

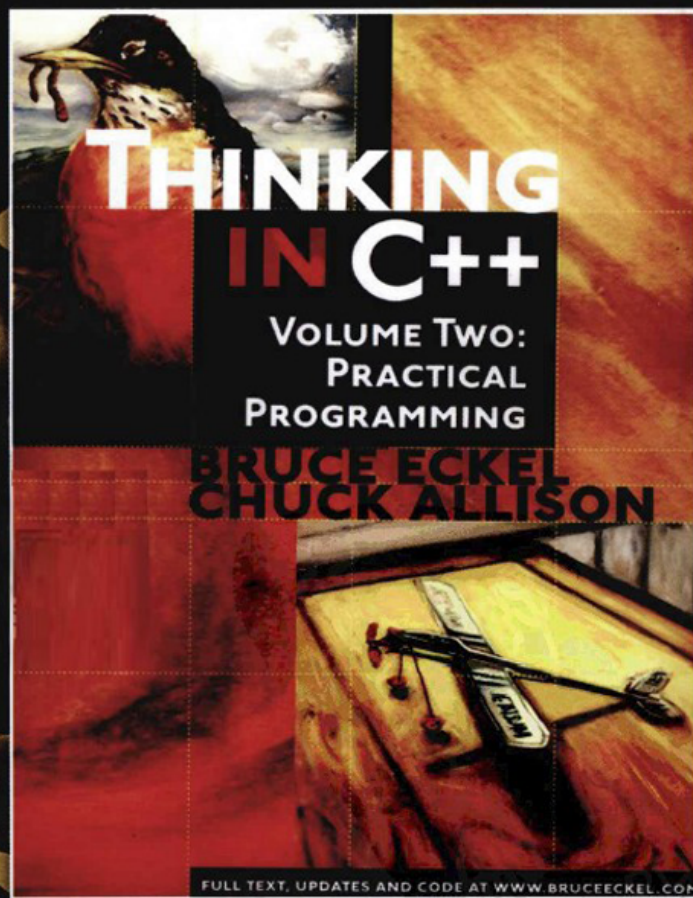
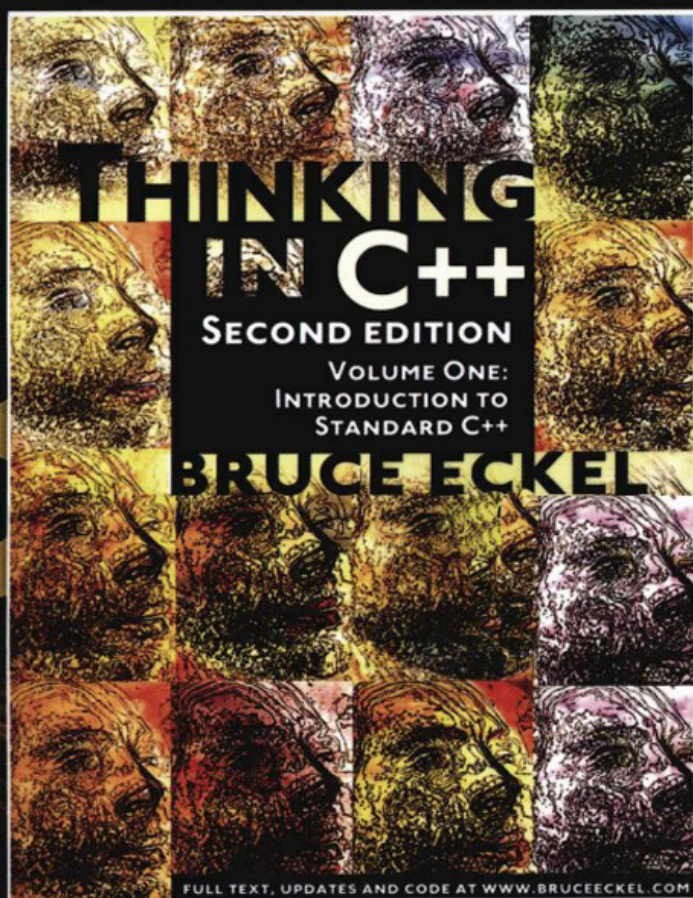
# C++编程思想

第1卷 标准C++导引 (原书第2版) &  
第2卷 实用编程技术

(美) Bruce Eckel Chuck Allison 著 刘宗田 袁兆山 潘秋菱 刁成嘉 等译

Thinking in C++

Volume One: Introduction to Standard C++, Second Edition  
& Volume Two: Practical Programming





# C++编程思想 (两卷合订本)

## Thinking in C++

Volume One: Introduction to Standard C++, Second Edition  
& Volume Two: Practical Programming

《C++编程思想》荣获《软件开发》杂志评选的1996年Jolt生产力大奖

本书是《C++编程思想》两卷的汇总。第1卷是在第1版的基础上进行了更加深入分析和修改后的第2版，其内容、讲授方法、选用实例以及配套的练习别具特色，可以供不同程度的读者选择阅读。第2卷介绍了C++实用的编程技术和最佳的实践方法，深入探究了异常处理方法和异常安全设计；介绍C++的字符串、输入输出流的现代用法；解释多重继承问题的难点，描述了典型的设计模式及其实现，特别介绍了多线程处理编程技术。

在本书作者的个人网站[www.BruceEckel.com](http://www.BruceEckel.com)上提供：

- 本书的英文原文、源代码、练习解答指南、勘误表及补充材料。
- 本书相关内容的研讨和咨询。
- 本书第1卷及第2卷英文电子版的免费下载链接。

本书的附录请到华章网站 ([www.hzbook.com](http://www.hzbook.com)) 下载。

### 作者简介

**Bruce Eckel** 是MindView公司的总裁，向客户提供软件咨询和培训。他是C++标准委员会拥有表决权的成员之一，他也是《Java编程思想》(该书第3版影印版及翻译版已由机械工业出版社引进出版)。他曾经写过另5本面向对象编程书籍，发表过150篇以上的文章，是多本计算机杂志的专栏作家。他经常参加世界各地的研讨会并进行演讲。



**Chuck Allison** 曾是《C/C++ Users》杂志的资深编辑，著有《C/C++ Code Capsules》一书。他是C++标准委员会的成员，犹他谷州立学院的计算机科学教授。他还是Fresh Sources公司的总裁，该公司专门从事软件培训和教学任务。



PEARSON

客服热线: (010) 88378991, 88361066  
购书热线: (010) 68326294, 88379649, 68995259  
投稿热线: (010) 88379604  
读者信箱: [hzjsj@hzbook.com](mailto:hzjsj@hzbook.com)

华章网站 <http://www.hzbook.com>

[www.pearsonhighered.com](http://www.pearsonhighered.com)

网上购书: [www.china-pub.com](http://www.china-pub.com)

封面设计: 金易 林

上架指导: 计算机 程序设计

ISBN 978-7-111-35021-7



9 787111 350217

定价: 116.00元



计 算 机 科 学 丛 书

两卷合订本

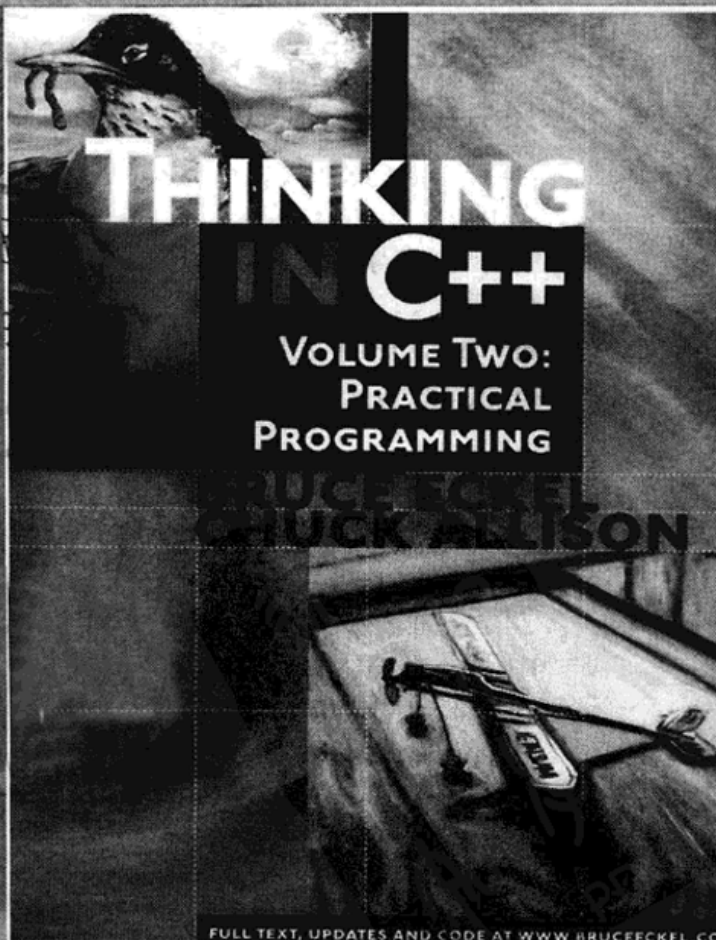
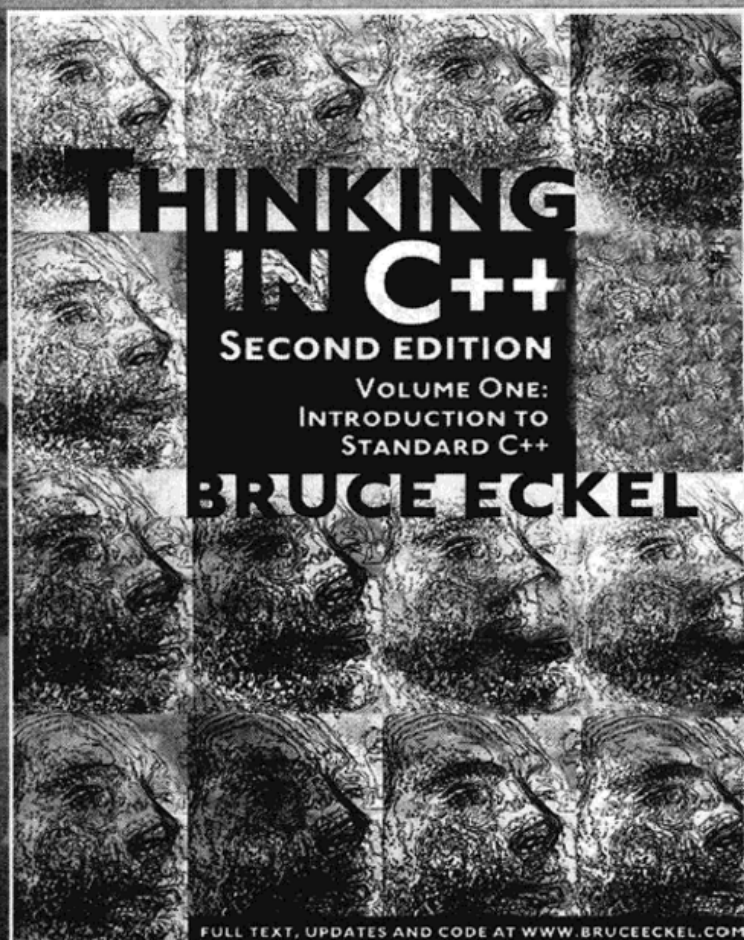
# C++编程思想

第1卷 标准C++导引 (原书第2版) &  
第2卷 实用编程技术

(美) Bruce Eckel Chuck Allison 著 刘宗田 袁兆山 潘秋菱 刁成嘉 等译

## Thinking in C++

Volume One: Introduction to Standard C++, Second Edition  
& Volume Two: Practical Programming



机械工业出版社  
China Machine Press



本书曾荣获美国《软件开发》杂志评选的1996年Jolt生产力大奖，中文版自2000年推出以来，经久不衰，获得了读者的充分肯定和高度评价。

本书的第1卷是在第1版的基础上进行了更加深入的分析和修改后得到的第2版，其内容更加集中，可以供不同程度的读者选择阅读。本书第2卷介绍了C++实用的编程技术和最佳的实践方法，深入探究了异常处理方法和异常安全设计；介绍C++的字符串、输入输出流的现代用法；解释多重继承问题的难点，描述了典型的设计模式及其实现，特别介绍了多线程处理编程技术。

本书是C++领域内一本权威的著作，书中的内容、讲授方法、练习既适合课堂教学，又适合读者自学。本书适合作为高等院校计算机及相关专业的本科生、研究生的教材，也可供从事软件开发的研究人员和科技工作者参考。

Authorized translation from the English language edition, entitled THINKING IN C++: INTRODUCTION TO STANDARD C++, VOLUME ONE, 2E, 9780139798092 by ECKEL, BRUCE, published by Pearson Education, Inc., publishing as Prentice Hall, Copyright © 1999 and THINKING IN C++, VOLUME Two: PRACTICAL PROGRAMMING, 1E, 0130353132, by ECKEL, BRUCE, ALLISON, CHUCK, Copyright © 2003.

All rights reserved.

CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD., and CHINA MACHINE PRESS Copyright © 2011.

本书封面贴有Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。

封底无防伪标均为盗版

版权所有，侵权必究

本书法律顾问 北京市展达律师事务所

本书版权登记号：图字：01-2011-3363

图书在版编目（CIP）数据

C++编程思想（两卷合订本）/（美）埃克尔（Eckel, B.）等著；刘宗田等译. —北京：机械工业出版社，2011.7

（计算机科学丛书）

书名原文：Thinking in C++: Volume One: Introduction to Standard C++, Second Edition & Volume Two: Practical Programming

ISBN 978-7-111-35021-7

I. C… II. ①埃… ②刘… III. C语言—程序设计 IV. TP312

中国版本图书馆CIP数据核字（2011）第110972号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）

责任编辑：秦 健

中国电影出版社印刷厂印刷

2011年7月第1版第1次印刷

185mm×260mm · 59印张

标准书号：ISBN 978-7-111-35021-7

定价：116.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：(010) 88378991；88361066

购书热线：(010) 68326294；88379649；68995259

投稿热线：(010) 88379604

读者信箱：hzjsj@hzbook.com



文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域中取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自1998年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与Pearson, McGraw-Hill, Elsevier, MIT, John Wiley & Sons, Cengage等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出Andrew S. Tanenbaum, Bjarne Stroustrup, Brian W. Kernighan, Dennis Ritchie, Jim Gray, Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, Abraham Silberschatz, William Stallings, Donald E. Knuth, John L. Hennessy, Larry L. Peterson等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专程为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方式如下：

华章网站：[www.hzbook.com](http://www.hzbook.com)

电子邮件：[hzjsj@hzbook.com](mailto:hzjsj@hzbook.com)

联系电话：(010) 88379604

联系地址：北京市西城区百万庄南街1号

邮政编码：100037



华章教育

华章科技图书出版中心



## 出版说明

Thinking in C++: Volume One: Introduction to Standard C++, Second Edition & Volume Two: Practical Programming

为更好地满足读者的不同需求，这次特别策划出版《C++编程思想（两卷合订本）》。根据读者对前一版本的意见，我们对图书进行了认真的更正和完善；同时，将光盘中的内容作为网络下载服务免费提供，以方便更多的学习者。此外，合订本的定价也将略低于单独购买两本图书的总价。





出版者的话

出版说明

## 第1卷 标准C++导引

译者序 .....	3
前言 .....	5
第1章 对象导言 .....	11
1.1 抽象的过程 .....	11
1.2 对象有一个接口 .....	12
1.3 实现的隐藏 .....	14
1.4 实现的重用 .....	15
1.5 继承：重用接口 .....	15
1.5.1 is-a 关系和is-like-a 关系 .....	18
1.6 具有多态性的可互换对象 .....	18
1.7 创建和销毁对象 .....	21
1.8 异常处理：应对错误 .....	22
1.9 分析和设计 .....	22
1.9.1 第0阶段：制定计划 .....	24
1.9.2 第1阶段：我们在做什么 .....	24
1.9.3 第2阶段：我们将如何建立对象 .....	26
1.9.4 第3阶段：创建核心 .....	28
1.9.5 第4阶段：迭代用例 .....	29
1.9.6 第5阶段：进化 .....	29
1.9.7 计划的回报 .....	30
1.10 极限编程 .....	30
1.10.1 先写测试 .....	31
1.10.2 结对编程 .....	32
1.11 为什么C++会成功 .....	32
1.11.1 一个较好的C .....	32
1.11.2 延续式的学习过程 .....	33
1.11.3 效率 .....	33
1.11.4 系统更容易表达和理解 .....	33

1.11.5 尽量使用库 .....	33
1.11.6 利用模板的源代码重用 .....	34
1.11.7 错误处理 .....	34
1.11.8 大型程序设计 .....	34
1.12 为向OOP转变而采取的策略 .....	34
1.12.1 指导方针 .....	35
1.12.2 管理的障碍 .....	35
1.13 小结 .....	37
第2章 对象的创建与使用 .....	38
2.1 语言的翻译过程 .....	38
2.1.1 解释器 .....	38
2.1.2 编译器 .....	39
2.1.3 编译过程 .....	39
2.2 分段编译工具 .....	40
2.2.1 声明与定义 .....	40
2.2.2 连接 .....	44
2.2.3 使用库文件 .....	44
2.3 编写第一个C++程序 .....	45
2.3.1 使用iostream类 .....	45
2.3.2 名字空间 .....	46
2.3.3 程序的基本结构 .....	47
2.3.4 “Hello, World!” .....	47
2.3.5 运行编译器 .....	48
2.4 关于输入输出流 .....	48
2.4.1 字符数组的拼接 .....	49
2.4.2 读取输入数据 .....	49
2.4.3 调用其他程序 .....	50
2.5 字符串简介 .....	50
2.6 文件的读写 .....	51
2.7 vector简介 .....	52
2.8 小结 .....	55
2.9 练习 .....	56

第3章 C++中的C .....	57	3.7.4 逻辑运算符 .....	85
3.1 创建函数 .....	57	3.7.5 位运算符 .....	85
3.1.1 函数的返回值 .....	58	3.7.6 移位运算符 .....	86
3.1.2 使用C的函数库 .....	59	3.7.7 一元运算符 .....	88
3.1.3 通过库管理器创建自己的库 .....	59	3.7.8 三元运算符 .....	88
3.2 执行控制语句 .....	60	3.7.9 逗号运算符 .....	89
3.2.1 真和假 .....	60	3.7.10 使用运算符时的常见问题 .....	89
3.2.2 <b>if-else</b> 语句 .....	60	3.7.11 转换运算符 .....	90
3.2.3 <b>while</b> 语句 .....	61	3.7.12 C++的显式转换 .....	90
3.2.4 <b>do-while</b> 语句 .....	61	3.7.13 <b>sizeof</b> ——独立运算符 .....	93
3.2.5 <b>for</b> 语句 .....	62	3.7.14 <b>asm</b> 关键字 .....	94
3.2.6 关键字 <b>break</b> 和 <b>continue</b> .....	63	3.7.15 显式运算符 .....	94
3.2.7 <b>switch</b> 语句 .....	64	3.8 创建复合类型 .....	94
3.2.8 使用和滥用 <b>goto</b> .....	65	3.8.1 用 <b>typedef</b> 命名别名 .....	95
3.2.9 递归 .....	65	3.8.2 用 <b>struct</b> 把变量结合在一起 .....	95
3.3 运算符简介 .....	66	3.8.3 用 <b>enum</b> 提高程度清晰度 .....	97
3.3.1 优先级 .....	66	3.8.4 用 <b>union</b> 节省内存 .....	98
3.3.2 自增和自减 .....	67	3.8.5 数组 .....	99
3.4 数据类型简介 .....	67	3.9 调试技巧 .....	106
3.4.1 基本内建类型 .....	67	3.9.1 调试标记 .....	106
3.4.2 <b>bool</b> 类型与 <b>true</b> 和 <b>false</b> .....	68	3.9.2 把变量和表达式转换成字符串 .....	108
3.4.3 说明符 .....	69	3.9.3 C语言 <b>assert()</b> 宏 .....	108
3.4.4 指针简介 .....	70	3.10 函数地址 .....	109
3.4.5 修改外部对象 .....	72	3.10.1 定义函数指针 .....	109
3.4.6 C++引用简介 .....	74	3.10.2 复杂的声明和定义 .....	109
3.4.7 用指针和引用作为修饰符 .....	75	3.10.3 使用函数指针 .....	110
3.5 作用域 .....	76	3.10.4 指向函数的指针数组 .....	111
3.5.1 实时定义变量 .....	77	3.11 <b>make</b> : 管理分段编译 .....	111
3.6 指定存储空间分配 .....	78	3.11.1 <b>make</b> 的行为 .....	112
3.6.1 全局变量 .....	78	3.11.2 本书中的 <b>makefile</b> .....	114
3.6.2 局部变量 .....	79	3.11.3 <b>makefile</b> 的一个例子 .....	114
3.6.3 静态变量 .....	80	3.12 小结 .....	116
3.6.4 外部变量 .....	81	3.13 练习 .....	116
3.6.5 常量 .....	82	第4章 数据抽象 .....	119
3.6.6 <b>volatile</b> 变量 .....	83	4.1 一个袖珍C库 .....	119
3.7 运算符及其使用 .....	83	4.1.1 动态存储分配 .....	122
3.7.1 赋值 .....	83	4.1.2 有害的猜测 .....	124
3.7.2 数学运算符 .....	83	4.2 哪儿出问题 .....	125
3.7.3 关系运算符 .....	85	4.3 基本对象 .....	126



4.4 什么是对象 .....	130	6.6 聚合初始化 .....	166
4.5 抽象数据类型 .....	131	6.7 默认构造函数 .....	168
4.6 对象细节 .....	131	6.8 小结 .....	169
4.7 头文件形式 .....	132	6.9 练习 .....	169
4.7.1 头文件的重要性 .....	132	第7章 函数重载与默认参数 .....	171
4.7.2 多次声明问题 .....	133	7.1 名字修饰 .....	172
4.7.3 预处理器指示 <b>#define</b> 、 <b>#ifdef</b> 和 <b>#endif</b> .....	134	7.1.1 用返回值重载 .....	172
4.7.4 头文件的标准 .....	134	7.1.2 类型安全连接 .....	172
4.7.5 头文件中的名字空间 .....	135	7.2 重载的例子 .....	173
4.7.6 在项目中使⤢用头文件 .....	135	7.3 联合 .....	176
4.8 嵌套结构 .....	136	7.4 默认参数 .....	178
4.8.1 全局作用域解析 .....	138	7.4.1 占位符参数 .....	179
4.9 小结 .....	139	7.5 选择重载还是默认参数 .....	180
4.10 练习 .....	139	7.6 小结 .....	183
第5章 隐藏实现 .....	142	7.7 练习 .....	183
5.1 设置限制 .....	142	第8章 常量 .....	185
5.2 C++的访问控制 .....	142	8.1 值替代 .....	185
5.2.1 <b>protected</b> 说明符 .....	144	8.1.1 头文件里的 <b>const</b> .....	186
5.3 友元 .....	144	8.1.2 <b>const</b> 的安全性 .....	186
5.3.1 嵌套友元 .....	146	8.1.3 聚合 .....	187
5.3.2 它是纯面向对象的吗 .....	148	8.1.4 与C语言的区别 .....	187
5.4 对象布局 .....	148	8.2 指针 .....	188
5.5 类 .....	149	8.2.1 指向 <b>const</b> 的指针 .....	189
5.5.1 用访问控制来修改 <b>Stash</b> .....	151	8.2.2 <b>const</b> 指针 .....	189
5.5.2 用访问控制来修改 <b>Stack</b> .....	151	8.2.3 赋值和类型检查 .....	190
5.6 句柄类 .....	152	8.3 函数参数和返回值 .....	191
5.6.1 隐藏实现 .....	152	8.3.1 传递 <b>const</b> 值 .....	191
5.6.2 减少重复编译 .....	152	8.3.2 返回 <b>const</b> 值 .....	191
5.7 小结 .....	154	8.3.3 传递和返回地址 .....	193
5.8 练习 .....	154	8.4 类 .....	195
第6章 初始化与清除 .....	156	8.4.1 类里的 <b>const</b> .....	196
6.1 用构造函数确保初始化 .....	156	8.4.2 编译期间类里的常量 .....	198
6.2 用析构函数确保清除 .....	157	8.4.3 <b>const</b> 对象和成员函数 .....	200
6.3 清除定义块 .....	159	8.5 <b>volatile</b> .....	204
6.3.1 <b>for</b> 循环 .....	160	8.6 小结 .....	205
6.3.2 内存分配 .....	161	8.7 练习 .....	205
6.4 带有构造函数和析构函数的 <b>Stash</b> .....	162	第9章 内联函数 .....	207
6.5 带有构造函数和析构函数的 <b>Stack</b> .....	164	9.1 预处理器的缺陷 .....	207

9.1.1 宏和访问 .....	209	11.3.1 按值传递和返回 .....	257
9.2 内联函数 .....	210	11.3.2 拷贝构造函数 .....	261
9.2.1 类内部的内联函数 .....	210	11.3.3 默认拷贝构造函数 .....	265
9.2.2 访问函数 .....	211	11.3.4 替代拷贝构造函数的方法 .....	266
9.3 带内联函数的 <b>Stash</b> 和 <b>Stack</b> .....	215	11.4 指向成员的指针 .....	267
9.4 内联函数和编译器 .....	218	11.4.1 函数 .....	269
9.4.1 限制 .....	219	11.5 小结 .....	271
9.4.2 向前引用 .....	219	11.6 练习 .....	271
9.4.3 在构造函数和析构函数里隐藏行为 .....	220	第12章 运算符重载 .....	274
9.5 减少混乱 .....	220	12.1 两个极端 .....	274
9.6 预处理器的更多特征 .....	221	12.2 语法 .....	274
9.6.1 标志粘贴 .....	222	12.3 可重载的运算符 .....	275
9.7 改进的错误检查 .....	222	12.3.1 一元运算符 .....	276
9.8 小结 .....	225	12.3.2 二元运算符 .....	279
9.9 练习 .....	225	12.3.3 参数和返回值 .....	288
第10章 名字控制 .....	227	12.3.4 不常用的运算符 .....	290
10.1 来自C语言中的静态元素 .....	227	12.3.5 不能重载的运算符 .....	295
10.1.1 函数内部的静态变量 .....	227	12.4 非成员运算符 .....	296
10.1.2 控制连接 .....	230	12.4.1 基本方针 .....	297
10.1.3 其他存储类型说明符 .....	232	12.5 重载赋值符 .....	297
10.2 名字空间 .....	232	12.5.1 <b>operator=</b> 的行为 .....	298
10.2.1 创建一个名字空间 .....	232	12.6 自动类型转换 .....	306
10.2.2 使用名字空间 .....	234	12.6.1 构造函数转换 .....	306
10.2.3 名字空间的使用 .....	237	12.6.2 运算符转换 .....	307
10.3 C++中的静态成员 .....	238	12.6.3 类型转换例子 .....	309
10.3.1 定义静态数据成员的存储 .....	238	12.6.4 自动类型转换的缺陷 .....	310
10.3.2 嵌套类和局部类 .....	241	12.7 小结 .....	312
10.3.3 静态成员函数 .....	242	12.8 练习 .....	312
10.4 静态初始化的相依性 .....	244	第13章 动态对象创建 .....	315
10.4.1 怎么办 .....	245	13.1 对象创建 .....	315
10.5 替代连接说明 .....	250	13.1.1 C从堆中获取存储单元的方法 .....	316
10.6 小结 .....	250	13.1.2 <b>operator new</b> .....	317
10.7 练习 .....	251	13.1.3 <b>operator delete</b> .....	317
第11章 引用和拷贝构造函数 .....	254	13.1.4 一个简单的例子 .....	318
11.1 C++中的指针 .....	254	13.1.5 内存管理的开销 .....	318
11.2 C++中的引用 .....	254	13.2 重新设计前面的例子 .....	319
11.2.1 函数中的引用 .....	255	13.2.1 使用 <b>delete void*</b> 可能会出错 .....	319
11.2.2 参数传递准则 .....	257	13.2.2 对指针的清除责任 .....	320
11.3 拷贝构造函数 .....	257	13.2.3 指针的 <b>Stash</b> .....	320



13.3 用于数组的new和delete .....	324	第15章 多态性和虚函数 .....	364
13.3.1 使指针更像数组 .....	325	15.1 C++程序员的演变 .....	364
13.4 耗尽内存 .....	325	15.2 向上类型转换 .....	365
13.5 重载new和delete .....	326	15.3 问题 .....	366
13.5.1 重载全局new和delete .....	327	15.3.1 函数调用捆绑 .....	366
13.5.2 对于一个类重载new和delete .....	328	15.4 虚函数 .....	366
13.5.3 为数组重载new和delete .....	330	15.4.1 扩展性 .....	367
13.5.4 构造函数调用 .....	332	15.5 C++如何实现晚捆绑 .....	369
13.5.5 定位new和delete .....	333	15.5.1 存放类型信息 .....	370
13.6 小结 .....	334	15.5.2 虚函数功能图示 .....	371
13.7 练习 .....	334	15.5.3 撩开面纱 .....	372
第14章 继承和组合 .....	336	15.5.4 安装vpointer .....	373
14.1 组合语法 .....	336	15.5.5 对象是不同的 .....	373
14.2 继承语法 .....	337	15.6 为什么需要虚函数 .....	374
14.3 构造函数的初始化表达式表 .....	339	15.7 抽象基类和纯虚函数 .....	375
14.3.1 成员对象初始化 .....	339	15.7.1 纯虚定义 .....	378
14.3.2 在初始化表达式表中的内建类型 .....	339	15.8 继承和VTABLE .....	378
14.4 组合和继承的联合 .....	340	15.8.1 对象切片 .....	380
14.4.1 构造函数和析构函数调用的次序 .....	341	15.9 重载和重新定义 .....	382
14.5 名字隐藏 .....	343	15.9.1 变量返回类型 .....	383
14.6 非自动继承的函数 .....	346	15.10 虚函数和构造函数 .....	385
14.6.1 继承和静态成员函数 .....	349	15.10.1 构造函数调用次序 .....	385
14.7 组合与继承的选择 .....	349	15.10.2 虚函数在构造函数中的行为 .....	386
14.7.1 子类型设置 .....	350	15.11 析构函数和虚拟析构函数 .....	386
14.7.2 私有继承 .....	352	15.11.1 纯虚析构函数 .....	388
14.8 protected .....	353	15.11.2 析构函数中的虚机制 .....	389
14.8.1 protected继承 .....	353	15.11.3 创建基于对象的继承 .....	390
14.9 运算符的重载与继承 .....	353	15.12 运算符重载 .....	392
14.10 多重继承 .....	355	15.13 向下类型转换 .....	394
14.11 渐增式开发 .....	355	15.14 小结 .....	396
14.12 向上类型转换 .....	356	15.15 练习 .....	397
14.12.1 为什么要“向上类型转换” .....	357	第16章 模板介绍 .....	400
14.12.2 向上类型转换和拷贝构造函数 .....	357	16.1 容器 .....	400
14.12.3 组合与继承（再论） .....	359	16.1.1 容器的需求 .....	401
14.12.4 指针和引用的向上类型转换 .....	360	16.2 模板综述 .....	402
14.12.5 危机 .....	360	16.2.1 模板方法 .....	403
14.13 小结 .....	361	16.3 模板语法 .....	404
14.14 练习 .....	361	16.3.1 非内联函数定义 .....	405
		16.3.2 作为模板的IntStack .....	406

16.3.3 模板中的常量	408
16.4 作为模板的Stash和Stack	409
16.4.1 模板化的指针Stash	411
16.5 打开和关闭所有权	415
16.6 以值存放对象	417
16.7 迭代器简介	418
16.7.1 带有迭代器的栈	425
16.7.2 带有迭代器的PStash	427
16.8 为什么使用迭代器	432
16.8.1 函数模板	434
16.9 小结	435
16.10 练习	435
附录A <sup>⊖</sup> 编码风格	
附录B <sup>⊖</sup> 编程准则	
附录C <sup>⊖</sup> 推荐读物	

## 第2卷 实用编程技术

译者序	441
前言	442

### 第一部分 建立稳定的系统

第1章 异常处理	448
1.1 传统的错误处理	448
1.2 抛出异常	450
1.3 捕获异常	451
1.3.1 try块	451
1.3.2 异常处理器	451
1.3.3 终止和恢复	452
1.4 异常匹配	453
1.4.1 捕获所有异常	454
1.4.2 重新抛出异常	454
1.4.3 不捕获异常	455
1.5 清理	456
1.5.1 资源管理	457
1.5.2 使所有事物都成为对象	458
1.5.3 auto_ptr	460
1.5.4 函数级的try块	461
1.6 标准异常	462

1.7 异常规格说明	464
1.7.1 更好的异常规格说明	467
1.7.2 异常规格说明和继承	467
1.7.3 什么时候不使用异常规格说明	468
1.8 异常安全	468
1.9 在编程中使用异常	471
1.9.1 什么时候避免异常	471
1.9.2 异常的典型应用	472
1.10 使用异常造成的开销	474
1.11 小结	476
1.12 练习	476
第2章 防御性编程	478
2.1 断言	480
2.2 一个简单的单元测试框架	482
2.2.1 自动测试	483
2.2.2 TestSuite框架	485
2.2.3 测试套件	488
2.2.4 测试框架的源代码	489
2.3 调试技术	493
2.3.1 用于代码跟踪的宏	494
2.3.2 跟踪文件	494
2.3.3 发现内存泄漏	495
2.4 小结	499
2.5 练习	500

### 第二部分 标准C++库

第3章 深入理解字符串	504
3.1 字符串的内部是什么	504
3.2 创建并初始化C++字符串	505
3.3 对字符串进行操作	508
3.3.1 追加、插入和连接字符串	508
3.3.2 替换字符串中的字符	509
3.3.3 使用非成员重载运算符连接	512
3.4 字符串的查找	513
3.4.1 反向查找	516
3.4.2 查找一组字符第1次或最后一次出现的位置	517
3.4.3 从字符串中删除字符	519
3.4.4 字符串的比较	520

⊖ 附录内容请到华章网站 (www.hzbook.com) 下载。

3.4.5 字符串和字符的特性 .....	523	5.1.2 默认模板参数 .....	582
3.5 字符串的应用 .....	527	5.1.3 模板类型的模板参数 .....	583
3.6 小结 .....	531	5.1.4 typename关键字 .....	587
3.7 练习 .....	531	5.1.5 以template关键字作为提示 .....	588
第4章 输入输出流 .....	534	5.1.6 成员模板 .....	589
4.1 为什么引入输入输出流 .....	534	5.2 有关函数模板的几个问题 .....	591
4.2 救助输入输出流 .....	537	5.2.1 函数模板参数的类型推断 .....	591
4.2.1 插入符和提取符 .....	537	5.2.2 函数模板重载 .....	594
4.2.2 通常用法 .....	540	5.2.3 以一个已生成的函数模板地址 作为参数 .....	595
4.2.3 按行输入 .....	541	5.2.4 将函数应用到STL序列容器中 .....	598
4.3 处理流错误 .....	542	5.2.5 函数模板的半有序 .....	600
4.4 文件输入输出流 .....	544	5.3 模板特化 .....	601
4.4.1 一个文件处理的例子 .....	544	5.3.1 显式特化 .....	601
4.4.2 打开模式 .....	546	5.3.2 半特化 .....	602
4.5 输入输出流缓冲 .....	546	5.3.3 一个实例 .....	604
4.6 在输入输出流中定位 .....	548	5.3.4 防止模板代码膨胀 .....	606
4.7 字符串输入输出流 .....	550	5.4 名称查找问题 .....	609
4.7.1 输入字符串流 .....	551	5.4.1 模板中的名称 .....	609
4.7.2 输出字符串流 .....	552	5.4.2 模板和友元 .....	613
4.8 输出流的格式化 .....	555	5.5 模板编程中的习语 .....	617
4.8.1 格式化标志 .....	555	5.5.1 特征 .....	617
4.8.2 格式化域 .....	556	5.5.2 策略 .....	621
4.8.3 宽度、填充和精度设置 .....	557	5.5.3 奇特的递归模板模式 .....	623
4.8.4 一个完整的例子 .....	557	5.6 模板元编程 .....	624
4.9 操纵算子 .....	560	5.6.1 编译时编程 .....	625
4.9.1 带参数的操纵算子 .....	560	5.6.2 表达式模板 .....	631
4.9.2 创建操纵算子 .....	562	5.7 模板编译模型 .....	636
4.9.3 效用算子 .....	563	5.7.1 包含模型 .....	636
4.10 输入输出流程序举例 .....	565	5.7.2 显式实例化 .....	637
4.10.1 维护类库的源代码 .....	565	5.7.3 分离模型 .....	638
4.10.2 检测编译器错误 .....	568	5.8 小结 .....	639
4.10.3 一个简单的数据记录器 .....	570	5.9 练习 .....	640
4.11 国际化 .....	573	第6章 通用算法 .....	642
4.11.1 宽字符流 .....	574	6.1 概述 .....	642
4.11.2 区域性字符流 .....	575	6.1.1 判定函数 .....	644
4.12 小结 .....	577	6.1.2 流迭代器 .....	646
4.13 练习 .....	577	6.1.3 算法复杂性 .....	647
第5章 深入理解模板 .....	580	6.2 函数对象 .....	648
5.1 模板参数 .....	580	6.2.1 函数对象的分类 .....	649
5.1.1 无类型模板参数 .....	580		



6.2.2	自动创建函数对象 .....	649
6.2.3	可调整的函数对象 .....	652
6.2.4	更多的函数对象例子 .....	653
6.2.5	函数指针适配器 .....	658
6.2.6	编写自己的函数对象适配器 .....	662
6.3	STL算法目录 .....	665
6.3.1	实例创建的支持工具 .....	666
6.3.2	填充和生成 .....	669
6.3.3	计数 .....	670
6.3.4	操作序列 .....	671
6.3.5	查找和替换 .....	674
6.3.6	比较范围 .....	679
6.3.7	删除元素 .....	681
6.3.8	对已排序的序列进行排序和运算 .....	684
6.3.9	堆运算 .....	691
6.3.10	对某一范围内的所有元素进行 运算.....	691
6.3.11	数值算法.....	697
6.3.12	通用实用程序 .....	699
6.4	创建自己的STL风格算法 .....	700
6.5	小结.....	701
6.6	练习.....	702
第7章	通用容器.....	706
7.1	容器和迭代器.....	706
7.2	概述.....	707
7.2.1	字符串容器 .....	711
7.2.2	从STL容器继承 .....	712
7.3	更多迭代器.....	714
7.3.1	可逆容器中的迭代器 .....	715
7.3.2	迭代器的种类 .....	716
7.3.3	预定义迭代器 .....	717
7.4	基本序列容器：vector、list和deque .....	721
7.4.1	基本序列容器的操作 .....	721
7.4.2	向量 .....	723
7.4.3	双端队列 .....	728
7.4.4	序列容器间的转换 .....	730
7.4.5	被检查的随机访问 .....	731
7.4.6	链表 .....	732
7.4.7	交换序列 .....	736
7.5	集合.....	737
7.6	堆栈.....	743
7.7	队列.....	745
7.8	优先队列.....	748
7.9	持有二进制位.....	755
7.9.1	bitset<n> .....	756
7.9.2	vector<bool> .....	758
7.10	关联式容器 .....	760
7.10.1	用于关联式容器的发生器和 填充器.....	763
7.10.2	不可思议的映像 .....	765
7.10.3	多重映像和重复的关键字 .....	766
7.10.4	多重集合 .....	768
7.11	将STL容器联合使用 .....	771
7.12	清除容器的指针 .....	773
7.13	创建自己的容器 .....	774
7.14	对STL的扩充 .....	776
7.15	非STL容器 .....	777
7.16	小结 .....	781
7.17	练习 .....	781
第三部分 专    题		
第8章	运行时类型识别 .....	785
8.1	运行时类型转换.....	785
8.2	typeid 操作符 .....	789
8.2.1	类型转换到中间层次类型 .....	790
8.2.2	void型指针 .....	791
8.2.3	运用带模板的RTTI .....	792
8.3	多重继承.....	793
8.4	合理使用RTTI .....	793
8.5	RTTI的机制和开销 .....	797
8.6	小结.....	797
8.7	练习.....	798
第9章	多重继承.....	800
9.1	概论.....	800
9.2	接口继承.....	801
9.3	实现继承.....	803
9.4	重复子对象.....	807
9.5	虚基类.....	810
9.6	名字查找问题.....	817
9.7	避免使用多重继承.....	819
9.8	扩充一个接口.....	820

9.9 小结	823	11.2 C++中的并发	876
9.10 练习	823	11.3 定义任务	878
第10章 设计模式	825	11.4 使用线程	879
10.1 模式的概念	825	11.4.1 创建有响应的用户界面	880
10.2 模式分类	826	11.4.2 使用执行器简化工作	882
10.3 简化习语	827	11.4.3 让步	884
10.3.1 信使	827	11.4.4 休眠	885
10.3.2 收集参数	828	11.4.5 优先权	886
10.4 单件	829	11.5 共享有限资源	887
10.5 命令：选择操作	833	11.5.1 保证对象的存在	887
10.6 消除对象耦合	836	11.5.2 不恰当地访问资源	890
10.6.1 代理模式：作为其他对象的前端	837	11.5.3 访问控制	892
10.6.2 状态模式：改变对象的行为	838	11.5.4 使用保护简化编码	893
10.7 适配器模式	840	11.5.5 线程本地存储	896
10.8 模板方法模式	841	11.6 终止任务	897
10.9 策略模式：运行时选择算法	842	11.6.1 防止输入/输出流冲突	897
10.10 职责链模式：尝试采用一系列策略模式	843	11.6.2 举例观赏植物园	898
10.11 工厂模式：封装对象的创建	845	11.6.3 阻塞时终止	901
10.11.1 多态工厂	847	11.6.4 中断	902
10.11.2 抽象工厂	849	11.7 线程间协作	906
10.11.3 虚构造函数	851	11.7.1 等待和信号	906
10.12 构建器模式：创建复杂对象	855	11.7.2 生产者-消费者关系	909
10.13 观察者模式	860	11.7.3 用队列解决线程处理的问题	912
10.13.1 “内部类”方法	862	11.7.4 广播	916
10.13.2 观察者模式举例	864	11.8 死锁	921
10.14 多重派遣	867	11.9 小结	925
10.15 小结	873	11.10 练习	926
10.16 练习	873		
第11章 并发	875		
11.1 动机	875		
		附录	
		附录A 推荐读物	
		附录B 其他	

## 附 录<sup>①</sup>

附录A 推荐读物

附录B 其他

① 附录内容请到华章网站 (www.hzbook.com) 下载。

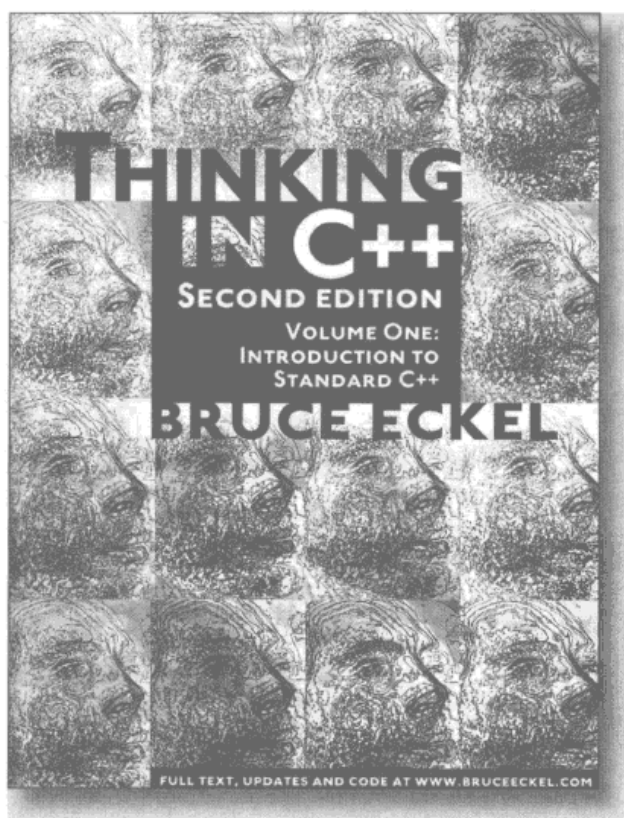


# 第1卷 标准C++导引

(原书第2版)

Volume One: Introduction to Standard C++  
Second Edition

(美) Bruce Eckel 著 刘宗田 袁兆山 潘秋菱 等译



本书为1996年软件开发图书Jolt生产力大奖得主。

“这本书是一项巨大的成就。你的书架上早就该有这本书了。讲述输入输出流的这一章是我至今见过的对这个主题最全面最易懂的叙述。”

——《Doctor Dobbs Journal》杂志的资深编辑Al Stevens

“Eckel的作品是惟一一本能如此清晰地叙述以面向对象方法构造程序的书籍。这本书也是一本优秀的C++入门指南。”

——《Unix Review》杂志的编辑Andrew Binstock

“Bruce继续用他对C++的洞察力让我惊叹。《C++编程思想》是他思想的精华荟萃。如果你需要C++中难题的清晰解答，就去买这部杰作吧。”

——《The Tao of Objects》一书的作者Gary Entsminger

“《C++编程思想》耐心而系统地探讨了C++的一些重要问题，例如内联、引用、运算符重载、继承和动态对象，还有一些更深奥的主题，例如模板的合理使用、异常和多继承等。所有这些内容，连同Eckel本人的关于对象和程序设计的观点，完美地编织为一体，成为每个C++开发者案头必备之书。如果你正在用C++开发软件，《C++编程思想》应当是你必须拥有的一本书。”

——《PC Magazine》的特邀编辑Richard Hale Shaw



作为译者，我有幸组织翻译了《C++编程思想》第1版。在这之前，我仅仅耳闻这是一本别具特色的畅销书，至于如何别具特色，如何得以畅销，并不十分清楚。在第1版的翻译过程中，我逐渐领悟了Eckel编写技巧的真谛。在第1版中文版的译者序中，我曾这样总结他的技巧：“其内容、讲授方法、选用例子和跟随的练习，别具特色。原书作者不是按传统的方法讲解C++的概念和编程方法，而是根据他自己过去学习C++的亲身体会，根据他多年教学中从他的学生们的学习中发现的问题，用一些非常简单的例子和简练的叙述，阐明了在学习C++中特别容易混淆的概念。特别是，他经常通过例子引导读者从C++编译实现的汇编代码的角度反向审视C++的语法和语义，常常使读者有‘心有灵犀一点通’的奇特效果，这在以往的C++书中并不多见。”

《C++编程思想》第1版的中文版自2000年1月第1次印刷以来，在中国市场上的畅销势头经久不衰。这充分说明了这本书在中国读者心目中的地位。

Eckel致力于计算机程序设计语言教学数十年，而且是全心全意地从事这项工作，这本身就是难能可贵的，是他成功的根本原因。另外，他的成功还有赖于他的精益求精的精神，这不仅表现在第1版的与众不同的精心选材和认真推敲的叙述方面，也体现在第2版与第1版的不同点上。

表面上，第2版与第1版并无太多的变化，但是通过分析，可以看出，其中的任何变化都是经过深思熟虑的。从章节上看，最大的区别是增加了两章和去掉了四章。增加的两章分别是“对象的创建与使用”和“C++中的C”。前者与“对象导言”实际上是第1版的“对象的演化”一章的彻底重写，增加了近几年面向对象方法和编程方法的最新研究与实践的丰硕成果。后者的添加不仅使不熟悉C的读者直接使用这本书成为可能，而且C本身就是C++的组成部分，这是C++得以成功的主要原因之一。删去的四章是“输入输出流介绍”、“多重继承”、“异常处理”和“运行时类型识别”。这四章属于C++中较复杂的主题，作者将它们连同C++标准完成后又增加的一些内容放到这本书的第2卷中。这样就使得这本书的第1卷内容更加集中，一般的读者可以不被这些复杂内容所困扰，而需要这些复杂知识的读者可以阅读这本书的第2卷。

实际上，第2版的改变不仅仅在于这些章节的调整，更多的改变体现在每一章中，包括例子的调整和练习的补充。这本书更成熟了。

受机械工业出版社华章公司计算机编辑部委托，我又承担起《C++编程思想》第2版的翻译组织任务。翻译这样的成功之作，既是机遇，更是压力。有如此众多的读者阅读我们翻译的作品，无论如何这是令人高兴的事情。诚然，吸引读者的魅力来源于原作，而不是我们的翻译技巧，但是能将如此光辉灿烂的作品变成中文版本，奉献给中国的读者，这其中毕竟融入了我们的心血，而且，第1版中文版的畅销，已经充分证明，并未因我们翻译水平的限制而黯淡了原作的光芒，这对我们已经够宽慰的了。然而，百万双眼睛在阅读这本书的同时也在审视我们的翻译水平，这就足以使我们诚惶诚恐的了。翻译在某种意义上是再创作的过程，



读者见仁见智。为求更多的满意，我们只有尽力而为。由于时间和水平限制，翻译错误在所难免，恳请读者指正。

参加第2版翻译和审校工作的人员包括：刘宗田、韩冬、蔡烈斌、袁兆山、潘秋菱、许东、李航、肖苑、刘璐、姜桂华、张卿、邵坤、陈慧琼、何允如、贾亮、童朝柱、邢大红、潘飏、刘莹、姜川、冯鸿等。

感谢为本书第1版和第2版中文版作出贡献的所有朋友。感谢关心和支持本书翻译出版的广大读者。

刘宗田



像任何人类语言一样，C++提供了一种表达思想的方法。如果这种表达方法是成功的，那么当问题变得更大和更复杂时，该方法将会明显地表现出比其他方法更容易和更灵活的优点。

不能只把C++看做是语言要素的一个集合，因为有些要素单独使用是没有意义的。如果我们不只是用C++语言编写代码，而是用它思考“设计”问题，那么必须综合使用这些要素。而且，为了以这种方法理解C++，我们必须了解使用C的问题和一般的编程问题。本书讨论的是编程问题、为什么这些编程问题会成为要解决的问题以及用C++解决编程问题所采用的方法。因此，在每一章中所解释的一组语言要素，都建立在C++语言解决某一类特殊问题所用方法的基础之上。以这种方式，我希望一点一点地引导读者，从掌握C开始，直到读者使用C++变成自己的母语思维方式。

我将始终坚持一种观点：读者应当在头脑中建立一个模型，以便从各个方面深入理解这门语言的精髓。如果读者遇到难题，他可以将问题纳入这个模型，推导出答案。我将努力把已经印在我脑海中的见解传授给读者，正是这些见解，使得我能开始“用C++进行思考”。

## 第1卷第2版中的新内容

本书是第1版的彻底重写，反映了C++标准最终完成所带来的C++的所有改变，也反映了自从第1版写完后我又学习到的内容。我已经检查并重写了第1版中的全部文字，在这个过程中，我删去了一些过时的例子，修改了一些现有的例子，并增加了一些新的例子和新的练习。我对第1版的内容进行了大规模的重新整理和重新编排，以便反映新出现的更好的工具和我对人们如何学习C++的进一步理解。为方便没有C背景知识的读者能阅读本书后面的章节，在第2版增加了一章，简要地介绍C概念和基本的C++特征。

因而，对于“第2版与第1版相比有何不同”这个问题的简要回答是：不同之处不在于版本号是新的，而是进行了重写，有的地方读者甚至无法认出原来的例子和材料。

## 第2卷的内容是什么

C++标准增加了一些重要的新库，例如String、在标准C++库中的容器和算法，以及模板中的新的复杂性。这些新增的内容和其他更高级的主题被放进本书的第2卷，包括多重继承、异常处理、设计模式和建立和调试稳定系统等内容。

## 如何得到第2卷

就像当前你手上的这本书一样，《C++编程思想》第2卷完全可以从我的网站[www.BruceEckel.com](http://www.BruceEckel.com)上下载。

## 预备知识

在本书第1版中，我假定读者已经学习了C，并至少具有自如阅读的水平。我的重点放在

简化我认为比较困难的部分：C++语言。第2版增加了一章，快速地介绍C，并在光盘上提供“Thinking in C”的课堂讨论材料，但是即使如此，我仍然假设读者具有一定的程序设计经验。另外，正如读者可以通过读小说而直接地学会许多新词一样，读者也可以从在本书后面的文字中学习有关于C的大量知识。

## 学习C++

我希望本书的读者有和我进入C++时相同的情况：作为一个C程序员，对于编程持有实在而执着的态度。但糟糕的是，我的背景和经验是在硬件层的嵌入式编程方面。在那里，C常常被看做高层语言，它对于位操作是低效率的。后来我发现，自己甚至不是一个好的C程序员，平时总是掩盖了对`malloc()`和`free()`、`setjmp()`和`longjmp()`结构以及其他“复杂”概念的无知，当开始触及这些主题时就竭力回避，而不是努力去获取新的知识。

在我开始致力于学习C++时，当时惟一像样的书是Stroustrup夫子自道式的“专家指南”<sup>①</sup>，因此我只好靠自己弄清基本概念。这引出了我的第一本关于C++的书<sup>②</sup>，这本书基本上就是直接把我头脑中的经验倒出来而写成的。它的构思是作为读者的指南，引导程序员同时进入C和C++。这本书的两个版本<sup>③</sup>都收到了读者的热情反响。

几乎就在《Using C++》出版的同时，我开始讲授这门语言。讲授C++已经变成了我的职业。自1989年以来，在授课时我看到了世界各地听众昏昏欲睡的样子、茫然不知的面容和困惑不解的表情。当我对一些人数不多的人群进行内部培训时，在练习过程中又发现了某些问题。即便那些面带微笑和会心点头的学生，实际上对许多问题也还是糊涂的。通过开创和多年主持“软件开发会议”的C++和Java系列专题，我发现，我和其他讲演者都有一种倾向，即过快地向听众灌输了过多的主题。后来，我做了一些努力，通过区别对待不同层次的听众和提供相关资料的方法，尽量吸引听众。也许这是过分的要求，但是因为我是一个抵触传统教学的人（对于大部分人而言，我相信这种抵触源于厌倦），所以希望我通过努力，使每一个人都能跟得上教学进度。

有一段时间，我编写了大量的教学演示。这样，我结束了通过实验和重复方式进行学习（在设计C++程序的过程中，这也是一项很有用的技术）的阶段。最后，从我多年的教学经验中总结出来的所有内容，形成了一门课程。在课程中，我用一系列分离的、易于理解的步骤并采用实地课堂讨论的形式解决学习中的问题（理想的学习情况），并在每次课后面跟随着练习。

本书的第1版是作为两学年制课程编写的，并且书中的内容已经在许多不同的课堂讨论上通过了多种形式的检验。我从每次课堂讨论上收集反馈意见，不断地修改和调整内容，直到我感觉到它已经成为一本很好的教材为止。但这本书不仅仅是课堂讨论的分发教材，而且我在其中放入了尽可能多的信息，在结构上使得它能引导读者顺利地通过当前主题和进入下一个主题。另外，这本书也适合于自学读者，能帮助他们尽快地掌握这门新的编程语言。

## 目标

在这本书中，我的目标是：

- 1) 以适当的进度介绍内容。每次将学习向前推进一小步，因此读者能很容易地在继续下

① Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986 (第1版).

② *Using C++*, Osborne/McGraw-Hill 1989.

③ *Using C++ and C++ Inside & Out*, Osborne/McGraw-Hill 1993.

一步学习之前消化每个已学过的概念。

2) 尽可能使用简短的例子。当然,这有时会妨碍我解决“现实世界”的问题。但是,我发现,当初学者能够掌握例子的每个细节,而不受问题的领域所影响时,他们通常会更有兴趣进行学习。另外,在课堂情况下能达到的接受能力,对代码的长短也有严格的限制。为此,我有时会受到使用“玩具例子”的批评,但是我甘愿承受这一批评,因为这样更有利于取得某些教学法上的效果。

3) 仔细安排描述内容的顺序,不让读者看到还没有揭示的内容。当然,这不是总能做到的;如果出现了这种情况,我将会给出简明的介绍性的描述。

4) 只把对于理解这门语言比较重要的东西介绍给读者,而不是介绍我知道的所有内容。我相信,不同信息的重要性是不同的。有些内容是95%的程序员不需要知道的,这些东西只会迷惑人们,增加他们对该语言复杂性的感觉。举一个C语言的例子,如果我们记住运算符优先表(我是记不住的),我们就可以写更漂亮的代码。但是,如果一定要这样做,反而会使代码的读者或维护者糊涂。所以可以忘掉优先级,当不清楚时使用括号。我们对于C++中的某些内容也可以采取同样的态度,因为我认为这些内容对于写编译器的人更重要,而对于程序员就不是那么重要。

5) 保持每一节的内容充分集中,使得授课时间以及两个练习之间的间隔时间不长。这不仅能使听众保持活跃的思想和在课堂讨论中精力集中,而且会使他们有更大的成就感。

6) 帮助读者打下坚实的基础,使得他们能充分地理解面对的问题,从而可以顺利地转向学习更困难的课程和书籍(特别是这本书的第2卷)。

7) 我尽力不用任何特定厂商的C++版本,因为对于学习编程语言,我不认为特定实现的细节像语言本身一样重要。大部分厂商的文档只适合于他们自己的特定实现。

## 各章概要

C++是一个在已有文法上面增加了新的不同特征的语言(因此,它被认为是混合的面向对象的编程语言)。由于很多人走了学习弯路,因此我们已经开始探索C程序员转移到C++语言特征的方法。因为这是过程型训练思想的自然延伸,所以我决定去理解和重复相同的道路,并通过引出和回答一些问题来加速这一进程,这些问题是当我学习该语言时遇到的和听众在听我的课时提出来的。

设计这门课时,我始终注意一件事:精练C++语言的学习过程。听众的反馈意见帮助我认识到哪些部分是很难学习的和需要额外解释的。在这个领域中,我曾经雄心勃勃,一次讲解包括了太多的内容。通过讲解过程,我知道了,如果包括大量新特征,就必须对它们全部作出解释,而且学生也特别容易混淆这些新特征。因此,我努力一次只介绍尽可能少的特征,理想的情况是每章一次只介绍一个主要概念。

本书的目标是只在每一章中讲授一个概念,或只讲授一小组相关的概念,用这种方法,不会依赖于其他的特征。这样,在进入下一章的学习之前,学生可以对自己的当前知识融会贯通。为了实现这个目标,我把一些C特征留到后面的章节去介绍,甚至放在比我希望的还要往后的地方介绍。这样做的好处是读者不会因为看到了许多未解释的C++特征被使用而困惑,因此,对该语言的介绍将是和缓的,并且将反映出读者自己消化这些特征时将会采用的方式。

下面是本书各章内容的简要说明。

**第1章 对象导言。**当项目对于维护而言变得太大和太复杂时,就产生了“软件危机”。



按程序员们的说法,“我们无法完成那些项目,即便能完成,它们也太昂贵了”。这引出了一些问题,在本章中我将讨论这些问题,并且讨论面向对象程序设计(OOP)的思想和如何运用这一思想解决软件危机问题。这一章引导读者了解OOP的基本概念和特征,介绍分析和设计过程。另外,在这一章中,我还将阐述采用这种语言的好处,提出关于如何转入C++语言领域的建议。

**第2章 对象的创建与使用。**这一章解释用编译器和库建立程序的过程。它介绍了本书中的第一个C++程序,显示程序如何构造和编译,然后介绍标准C++中的可用对象的基本库。在结束这一章时,我们就对如何用流行的对象库编写C++程序有一个深刻的领会。

**第3章 C++中的C。**这一章详细综述在C++中使用的C的特征和一些只在C++中使用的特征,还介绍在软件开发领域通用的“制作”工具,并且用它建立了本书中的所有例子(本书的源代码在[www.BruceEckel.com](http://www.BruceEckel.com)中可找到,包含了对每章的makefile)。第3章假设读者已经具有某种过程型程序设计语言的坚实基础,例如Pascal和C语言或者甚至某种形式的Basic(只要读者已经用这种语言编写了大量的代码,特别是函数)。

**第4章 数据抽象。**C++的大部分特征都围绕着创建新数据类型的能力。这不仅可以提供优质代码组织,而且可以为更强大的OOP能力奠定基础。读者将可以看到如何用将函数放入结构内部的简单过程来实现这一思想,并可以看到如何具体地完成这样的过程和创建什么样的代码。读者还能学会组织代码成为头文件和实现文件的最好方法。

**第5章 隐藏实现。**通过说明结构中的一些数据和函数是**private**(私有的),可以把它们设置为对于这个新结构类型的用户是不可见的。这意味着能够把下层实现和客户程序员看到的接口隔离开来,这样就容易改变具体实现,而不影响客户代码。另外,C++还引入关键字**class**作为描述新数据类型的更具吸引力的方法,而单词“对象”的意思并不神秘,它是一种美妙的变量。

**第6章 初始化与清除。**C语言的最通常的一类错误是由于变量未初始化而引起的。C++的构造函数使得程序员能保证他的新数据类型(即“他的类的对象”)的变量总是能被恰当地初始化。如果他的对象还需要某种方式的清除,他可以保证这个清除动作总是由C++的析构函数来完成。

**第7章 函数重载与默认参数。**C++可以帮助程序员建立大而复杂的项目。这时,可能会引进使用相同函数名的多个库,还可能会在同一个库中选择具有不同含义的相同的名字。C++采用函数重载使这一问题容易解决。重载允许当参数表不同时重用相同的函数名。默认参数通过自动为某些参数提供默认值,使我们能用不同的方式调用同一个函数。

**第8章 常量。**本章讨论了**const**和**volatile**关键字,它们在C++中有另外的含义,特别是在类的内部。我们将学习对指针定义使用**const**的含义。本章还说明**const**的含义在类的内部和外部有何不同,以及如何在类的内部创建编译时常量。

**第9章 内联函数。**预处理宏省去了函数调用开销,但是也排除了有价值的C++类型检查。内联函数具有预处理宏和实际函数调用的所有好处。这一章深入地研究了内联函数的实现和使用。

**第10章 名字控制。**在程序设计中,创建名字是基本的活动,而当项目变大时,名字的数量是无法限制的。C++允许在名字创建、可视性、存储代换和连接方面有大量的控制。这一章将说明如何在C++中用两种技术控制名字。第一,用关键字**static**控制可视性和连接,研究它对于类的特殊含义。另一种在全局范围内更有用的控制名字的技术是C++的**namespace**

(名字空间)特征,它允许把全局名字空间划分为不同的区域。

**第11章 引用和拷贝构造函数。**C++指针的作用和C指针一样,而且具有更强的C++类型检查的好处。C++还提供了另外的处理地址的方法:继Algol和Pascal之后,C++使用了“引用”,允许当程序员使用平常的符号时由编译器来处理地址操作。读者还会遇到拷贝构造函数,它通过按值传送控制将对象传送给函数或从函数中返回的方式。最后,本章还将解释C++指向成员的指针。

**第12章 运算符重载。**这个特征有时被称为“语法糖 (syntactic sugar)”。由于运算符也可以像函数调用那样使用,这使得程序员在运用类型的语法时更加灵活。在这一章中,读者将了解到,运算符重载只是不同类型的函数调用,并且将学会如何写自己的运算符重载,学会处理参数、类型返回以及确定一个运算符是成员还是友元时的一些易混淆的情况。

**第13章 动态对象创建。**一个空中交通系统将处理多少架飞机?一个CAD系统将需要多少种造型?在一般的程序设计问题中,我们不可能知道程序运行所需要的对象的数量、生命周期和类型。在这一章中,我们将学习C++的**new**和**delete**如何漂亮地通过在堆上安全地创建对象而解决上述问题。我们还将看到,**new**和**delete**如何用不同的方法重载,从而使我们能控制如何分配和释放存储。

**第14章 继承和组合。**数据抽象允许程序员从零开始创建新的类型。通过组合和继承,程序员可以用已存在的类型创建新的类型。用组合方法,程序员可以以老的类型作为零件组装成新的类型;而用继承方法,程序员可以创建已存在类型的一个更特殊的版本。在这一章中,读者将学习这一文法,学习如何重定义函数,以及理解构造和析构对于继承和组合的重要性。

**第15章 多态性和虚函数。**单靠读者自己,可能要花九个月的时间才能发现和理解OOP的这一基石。通过一些小而简单的例子,读者可以看到如何通过继承创建一个类型族,看到在这个类型族中如何通过公共基类对这个族中的对象进行操作。关键字**virtual**使程序员能够按类属处理这个族中的所有对象,这意味着大块代码将不依赖于特殊类型的信息,因此,程序是可扩充的,构造程序和维护代码也变得更加容易和更便宜。

**第16章 模板介绍。**继承和组合允许程序员重用对象代码,但不能解决有关重用需要的所有问题。模板通过为编译器提供了一种在类或函数体中代换类型名的方法,来允许程序员重用源代码。这就支持了容器类库的使用,容器类库是使我们能快速而有效地开发面向对象程序的重要工具(标准C++库包含了一个重要的容器类库)。这一章给出了这个基本主题的详尽阐述。

另一些主题(更高级的课题),可以在本书的第2卷中看到,本书的第2卷可以从网站[www.BruceEckel.com](http://www.BruceEckel.com)下载。

## 练习

我已经发现,在课堂讨论期间,练习对同学们的完全理解是特别有用的,因此,本书的每章后面都有一组练习。练习题的数量在第1版的基础上有很大的增加。

在这些练习中,很多是比较简单的,可以在课堂内或实验室内用合理的时间完成,老师可以通过观察证实所有的学生正在吸收这些材料。有些练习是为了激发优秀学生的学习兴趣。大量练习被设计成能在短期内求解,目的是为了测试和完善学生所掌握的知识,而不是提出主要的挑战(也许我们将自己找到那些挑战,更可能的是,那些挑战会自动出现在我们面前)。

## 练习的答案

部分练习题的答案可以在本书的电子文档“*Thinking in C++ Annotated Solution Guide*”中找到，只需支付很少的费用就可以从网站[www.BruceEckel.com](http://www.BruceEckel.com)上获得这个电子文档。

## 源代码

本书中的源代码是免费软件版权，通过网站[www.BruceEckel.com](http://www.BruceEckel.com)分发。该版权防止未经许可用印刷媒体重印这些代码，但是，在许多其他情况下可以使用这些代码。

这些代码放在一个压缩文件中，可以从任何有zip工具的平台提取（如果没有安装合适的平台，可以从Internet上找到适合你的平台的某个版本）。

读者可以在自己的项目中和在课堂上使用这些代码，只要遵守代码中的版权声明。

## 语言标准

在本书中，当谈到遵循ISO C标准时，我一般只是说‘C’。只有当有必要区别标准C和老的、以前版本的C时，我才加以区分。

在写这本书时，C++标准委员会完成了语言的标准化工作。这样，我将用术语“标准C++”来指代这个标准化的语言。如果我简单地谈到C++，读者就应该假设这意味着“标准C++”。

在C++标准委员会的实际名字与标准本身的名字之间有些混淆。委员会的主席Steve Clamage就此作了如下澄清：

有两个C++标准委员会：NCITS（以前的X3）J16委员会和ISO JTC1/SC22/WG14委员会。ANSI授权NCITS建立制订美国国家标准的技术委员会。

1989年J16受委托制订C++美国标准。1991年WG14受委托制订国际标准。J16项目转变为“Type I”（国际）项目，并服从于ISO标准化计划。

这两个委员会在同一时间、同一地点开会，J16的投票作为美国在WG14的票数。WG14委派J16做技术工作，并对J16的技术工作进行表决。

最初，C++标准是作为ISO标准制订的。ANSI后来投票（在J16的建议下）决定采用ISO C++标准作为C++美国标准。

因此，“ISO”是称呼C++标准的正确方式。

## 语言支持

读者的编译器可能不支持在本书中讨论的所有特征，如果还没有编译器的最新版本就更是如此了。实现像C++这样的语言是艰巨的任务，而且读者可能希望将这些特征划分成一些部分后分别出现，而不是一下子全出现。如果读者试用本书中的某个例子，从编译器中得到了大量的错误，这可能不是代码或编译错误，而可能是他的特定编译器还没有实现例子中的某个特征。

## 错误

无论一个作者具有多少发现错误的技巧，总会有一些错误漏网，它们常常能被新读者发现。如果读者发现了任何认为是错误的地方，请填写网站[www.BruceEckel.com](http://www.BruceEckel.com)上关于本书的修改表格并在线提交，我将非常感激。

## 对象导言

计算机革命起源于一台机器。因此，程序设计语言的起源看上去也起源于那台机器。

然而，计算机并不仅仅是一台机器，因为它们就像是智力放大工具（Steve Jobs<sup>①</sup>喜欢称其为“智力的自行车”）和另一种富于表现力的媒体。所以，它们渐渐地不太像机器，而更像我们大脑的一部分了，它们还挺像其他的一些富于表现力的媒体（例如写作、绘画、雕刻、动画或电影制作）。面向对象程序设计是“使计算机成为一种富于表现力的媒体”这一华彩乐章的一部分。

本章将介绍面向对象程序设计（OOP）的基本概念，包括OOP开发方法的概述。在读者阅读本书之前，我们假设读者已经有了使用过程型程序设计语言的经验，当然不一定是C语言。

本章是一些背景和辅助材料。如果在未了解这些大背景之前就进入面向对象程序设计，许多人都会感到有困难。因此，这里为读者预先介绍有关OOP的一些基本概念。但是，还有另一些读者，在不见到某些具体结构之前，就不能理解语言的整体概念；这些人不接触某些代码就会停止不前和不知所措。如果读者属于后者，急于学习这门语言的具体内容，则可以跳过本章，这样不会妨碍写程序和学习这门语言。但是，读者以后终将回过头来补充本章的知识，这样才能理解为什么对象如此重要，以及如何用对象设计程序。

### 1.1 抽象的过程

所有的程序设计语言都提供抽象。可以说，人们能解决的问题的复杂性直接与抽象的类型和质量有关。这里“类型”指的是“要抽象的东西”。汇编语言是对底层机器的小幅度抽象。其后的许多所谓“命令式”语言（例如Fortran、BASIC和C）都是对汇编语言的抽象。这些语言较之汇编语言有了很大的改进，但是它们的主要抽象仍然要求程序员按计算机的结构去思考，而不是按要解决的问题的结构去思考。程序员必须在机器模型（在“解空间”，即建模该问题的空间中，例如在计算机中）和实际上要解决的问题的模型（在“问题空间”，即问题存在的空间中）之间建立联系。程序员必须努力进行这之间的对应，这实际上是程序设计语言之外的任务，它使得程序难以编写且维护费用昂贵，并且作为一种副效应，造就了整个“程序设计方法”产业。

取代对机器建模的另一种方式是对要解决的问题进行建模。像LISP和APL这样的早期语言选取了特殊的“世界观”（“所有的问题最终都是表”或“所有的问题都是算法”），PROLOG则把所有的问题都看做决策链。还有一些语言，创造它们的目的是为了基于约束的编程和专门用于通过绘图符号来编程（后者已被证明局限太大）。这些方法中的每一种对于它所针对的特定问题都是很好的解决方案，但对于这个领域之外的问题，就笨拙难用了。

---

① 史蒂夫·乔布斯（Steve Jobs）是苹果公司的创始人和精神领袖。——编辑注



面向对象的方法为程序员提供了在问题空间中表示各种事物元素的工具；从而向前迈进了一大步。这种表示方法是通用的，并不限定程序员只处理特定类型的问题。我们把问题空间中的事物和它们在解空间中的表示称为“对象”（当然，还需要另外一些对象，它们在问题空间中并没有对应物）。其思想是允许程序通过添加新的对象类型，而使程序本身能够根据问题的实际情况进行调整，这样当我们读描述解决方案的代码时，也就是在读表达该问题的文字。这是比以前更灵活和更强大的语言抽象。因此，OOP允许程序员用问题本身的术语来描述问题，而不是用要运行解决方案的计算机的术语来描述问题。当然这些问题的术语仍然与计算机有联系。每个对象看上去像一台小计算机，它有状态，有可以执行的运算。这似乎是现实世界中对象的很好类比，它们都有特性和行为。

一些语言的设计者认为，面向对象程序设计本身并不足以轻易解决所有程序设计问题，他们提倡结合各种方法，形成多范型（*multiparadigm*）的程序设计语言<sup>①</sup>。

Alan Kay总结了Smalltalk的五个基本特性，这些特性代表了纯面向对象程序设计的方法。Smalltalk是第一个成功的面向对象语言，是C++的基础语言之一。

(1) **万物皆对象**。对象可以被认为是一个奇特的变量，它能存放数据，而且可以对它“提出请求”，要求它执行对它自身的运算。理论上，我们可以在需要解决的问题中取出任意概念性的成分（狗、建筑物、服务等），把它表示为程序中的对象。

(2) **程序就是一组对象**，对象之间通过发送消息互相通知做什么。更具体地讲，可以将消息看做是对于调用某个特定对象所属函数的请求。

(3) **每一个对象都有它自己的由其他对象构成的存储区**。这样，就可以通过包含已经存在的对象创造新对象。因此，程序员可以构造出复杂的程序，而且能将程序的复杂性隐藏在对象的简明性背后。

(4) **每个对象都有一个类型**。采用OOP术语，每个对象都是某个类的实例（instance），其中“类”（class）与“类型”（type）是同义词。类的最重要的突出特征是“能向它发送什么消息”。

(5) **一个特定类型的所有对象都能接收相同的消息**。正如后面将看到的，这种特性具有丰富的含义。因为一个“circle”类型的对象也是一个“shape”类型的对象，所以保证circle能接收shape消息。这意味着，我们可以编写与shape通信的代码，该代码能自动地对符合shape描述的任何东西进行处理。这种替换能力（*substitutability*）是OOP的最强大的思想之一。

## 1.2 对象有一个接口

亚里士多德可能是第一个认真研究类型（*type*）概念的人，他提到了“鱼类和鸟类”。所有对象（虽然都具有惟一性）都是一类对象中的一员，它们有共同的特征和行为。这一思想在第一个面向对象语言Simula-67中得到了直接的应用，该语言用基本关键字**class**在程序中引入新类型。

顾名思义，创造Simula的目的是为了解决模拟问题，例如著名的“银行出纳员问题”<sup>②</sup>。其中包括出纳员、顾客、账户、交易、货币的单位等大量的“对象”。把那些在程序执行期间的状态之外其他方面都一样的对象归为“几类对象”，这就是关键字**class**（类）的来源。创建抽象数据类型是面向对象程序设计的基本思想。抽象数据类型几乎能完全像内建类型一样工

① 参见Timothy Budd所写的专著《*Multiparadigm Programming in Leda*》(Addison-Wesley, 1995)。

② 可以在本书的第2卷中找到这一问题的有趣实现，本书的第2卷可从[www.BruceEckel.com](http://www.BruceEckel.com)上找到。

作。程序员可以创建类型的变量[面向对象的说法称为对象(*object*)或实例(*instance*)]和操纵这些变量(称为发送消息或请求,对象根据发来的消息推断需要做什么事情)。每个类的成员(元素)都有共性:每个账户有余额,每个出纳员都能接收存款,等等。同时,每个成员都有自己的状态,每个账户有不同的余额,每个出纳员都有名字。这样,在计算机程序中,出纳员、客户、账户、交易等,每一个都被描述为惟一的实体。这个实体就是对象,每个对象都属于一个定义了它的特性和行为的特定类。

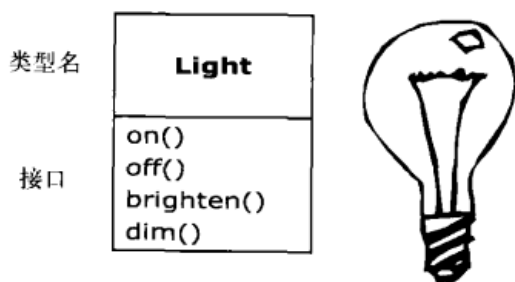
所以,虽然在面向对象的程序设计中,我们所做的工作实际上是创造新数据类型,但事实上所有的面向对象的程序设计语言都使用关键字“class”。当碰到“类型(*type*)”时可以看做“类(*class*)”,反之亦然<sup>①</sup>。

由于类描述了一组有相同特性(数据元素)和相同行为(功能)的对象,因此类实际上就是数据类型,例如浮点数也有一组特性和行为。区别在于,程序员定义类是为了与具体问题相适应,而不是被迫使用已存在的数据类型,而设计这些已存在的数据类型的动机是为了表示机器中的存储单元。程序员可以通过增添专门针对自己需要的新数据类型来扩展程序设计语言。这种程序设计系统欢迎新的类,关注新的类,对它们进行与内置类型一样的类型检查。

面向对象方法并不限于模拟创建。无论我们是否同意,都不能否认任何程序都是我们正在设计的系统的一种模拟,OOP技术的使用确实可以容易地将大量问题缩减为一个简单的解决方案。

一旦建立了一个类,程序员想制造这个类的多少个对象就可以制造多少个,然后操作这些对象,就如同它们是所解决的问题中的元素。实际上,面向对象程序设计的难题之一,是在问题空间中的元素和解空间的对象之间建立一对一的映射。

但是,我们如何得到一个对象去为我们做有用工作呢?必须有一种方法能向对象作出请求,使得它能做某件事情,例如完成交易、在屏幕上画图或打开开关。每个对象只能满足特定的请求。可以向对象发出的请求是由它的接口(*interface*)定义的,而接口由类型确定。一个简单的例子可能该是电灯泡的表示了。



```
Light lt;
lt.on();
```

接口规定我们能向特定的对象发出什么请求。然而,必须有代码满足这种请求,再加上隐藏的数据,就组成了实现(*implementation*)。从过程型程序设计的观点看,这并不复杂。类型对每个可能的请求都有一个相关的函数,当向对象发请求时,就调用这个函数。这个过程通常概括为向对象“发送消息”(提出请求),对象根据这个消息确定做什么(执行代码)。

如上例,这个类型或称类的名字是**Light**,这个特定的**Light**对象的名字是**lt**,可以对**Light**对象提出一些请求:打开它、关闭它、使它变亮或变暗。通过声明一个名字(**lt**),可以

<sup>①</sup> 一些人对此做了区分,他们认为类型(*type*)确定接口,而类(*class*)是对这个接口的特定实现。

创建一个**Light**对象。为了向这个对象发送消息，可以说出这个对象的名字，并用句点连接对消息的请求。从使用预定义类的用户的角度看，用对象编程只需要这些工作就行了。

上图符合统一建模语言（Unified Modeling Language, UML）的格式，每个类由一个方框表示，这个方框的顶部标有类型名，中间部分列出所关注的成员，底部是成员函数（member function属于这个对象的函数，它们能接收发送给这个对象的任何消息）。通常，只有类的名字和公共成员函数会在UML设计图中表示出来，所以中间部分不显示。如果只对类的名字感兴趣，则底部也不需要显示。

### 1.3 实现的隐藏

把程序员划分为类创建者（创建新数据类型的人）和客户程序员<sup>①</sup>（在应用程序中使用数据类型的类的用户）是有益的。客户程序员的目标是去收集一个装满类的工具箱，用于快速应用开发。类创建者的目标是去建造类，这个类只暴露对于客户程序员是必需的东西，其他的都隐藏起来。为什么呢？因为如果是隐藏的东西，客户程序员就不能使用它，这意味着，这个类的创建者可以改变隐藏的部分，而不用担心会影响其他人。被隐藏的部分通常是对象内部的管理功能，容易受到粗心或无知的客户程序的损害，所以隐藏实现会减少程序错误。隐藏的概念怎么强调都不过分。

在任何关系中，确定一个所有的参与者都遵从的边界是重要的。当我们创建一个库时，也就与客户程序员建立了关系。他们也是程序员，但是他们是通过使用我们的库来组装应用程序，也可能建立更大的库。

如果类的所有成员对于任何人都能用，那么客户程序员就可以用这个类做其中的任何事情，不存在强制规则。虽然我们可能实际上不希望客户程序员直接操纵这个类的某些成员，但是因为没有访问控制，所以就没有办法保护它，所有的东西都暴露无遗。

访问控制的第一个理由是为了防止客户程序员插手他们不应当接触的部分，也就是对于数据类型的内部实施方案是必需的部分，而不是用户为了解决他们的特定问题所需要的接口部分。这实际上是对用户的很好服务，因为这使得他们容易看到哪些部分对于他们是重要的，哪些部分是可以忽略的。

访问控制的第二个理由是允许库设计者去改变这个类的内部工作方式，而不必担心这样做会影响客户程序员。例如，库设计者可能为了容易开发而用简单的方法实现了一个特殊的类，但后来发现需要重写这个类以使得它运行得更快。如果接口和实现是严格分离和被保护的，那么库设计者就可以很容易完成重写任务，用户只需要重新连接就可以了。

C++语言使用了三个明确的关键字来设置类中的边界：**public**、**private**和**protected**。它们的使用和含义相当简明。这些访问说明符（*access specifier*）确定谁能用随后的定义。**public**意味着随后的定义对所有人都可用。相反，**private**关键字则意味着，除了该类型的创建者和该类型的内部成员函数之外，任何人都不能访问这些定义。**private**在我们与客户程序员之间筑起了一道墙。如果有人试图访问一个私有成员，就会产生一个编译时错误。**protected**与**private**基本相似，只有一点不同，即继承的类可以访问**protected**成员，但不能访问**private**成员。我们将在稍后部分介绍继承。

① 对于这个术语，我要感谢我的朋友Scott Meyers（名著《Effective C++》的作者——编辑注）。

## 1.4 实现的重用

创建了一个类并进行了测试后，这个类（理论上）就成了有用的代码单元。重用性并不像许多人所希望的那样容易达到，产生一个好设计需要经验和洞察力。但是只要有了这样的设计，就应该提供重用。代码重用是面向对象程序设计语言的最大优点之一。

重用一个类最简单的方法就是直接使用这个类的对象，并且还可以将这个类的对象放到一个新类的里面。我们称之为“创建一个成员对象”。可以用任何数量和类型的其他对象组成新类，通过组合得到新类所希望的功能。因为这是由已经存在的类组成新类，所以称为组合（*composition*）[或者更通常地称为聚合（*aggregation*）]。组合常常被称为“has-a（有）”关系，比如“在小汽车中有发动机”一样。



（上面的UML图用实心菱形表示组合，说明这里有一个小汽车。通常我将采用一种更简单的形式，即仅用一条不带菱形的线来表示一个关联。<sup>①</sup>）

组合带来很大的灵活性，新类的成员对象通常是私有的，使用这个类的客户程序员不能访问它们。这种特点允许我们改变这些成员而不会干扰已存在的客户代码。我们还可以在运行时改变这些成员对象，动态地改变程序的行为。下面将介绍的继承没有这种灵活性，因为编译器必须在用继承方法创造的类上加入编译时限制。

因为继承在面向对象的程序设计中很重要，所以它常常得到高度重视，并且新程序员可能会产生在任何地方都使用继承的想法。这会形成拙笨和极度复杂的设计。实际上，当创建新类时，程序员应当首先考虑组合，因为它更简单和更灵活。如果采用组合的方法，设计将变得清晰。一旦我们具备一些经验之后，就能很明显地知道什么时候需要采用继承方法。

## 1.5 继承：重用接口

对象的思想本身是一种很方便的工具。它允许我们将数据和功能通过概念封装在一起，使得我们能描述合适的问题空间思想，而不是被强制使用底层机器的用语。通过使用class关键字，这些概念被表示为程序设计语言中的基本单元。

然而，克服许多困难去创建一个类，并随后强制性地创建一个有类似功能的全新的类，似乎并不是一种很好的方法。如果能选取已存在的类，克隆它，然后对这个克隆增加和修改，则是再好不过的事。这是继承（*inheritance*）带来的好处，例外的情况是，如果原来的类（称为基类、超类或父类）被修改，则这个修改过的“克隆”（称为派生类、继承类或子类）也会表现出这些改变。



① 这种形式对于大部分图通常是足够详细的，不需要特别指出何处使用聚合或称组合。

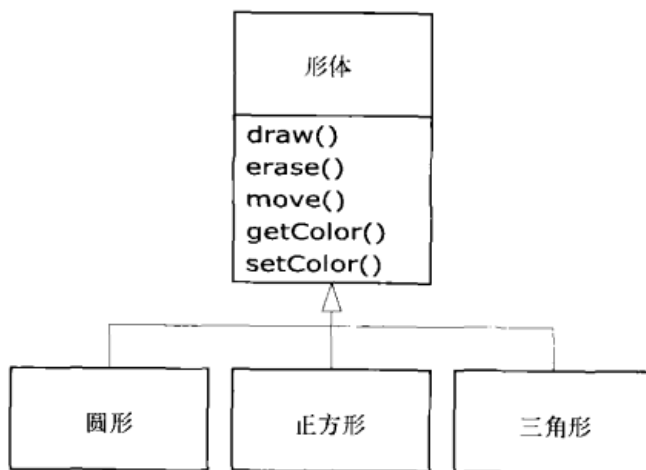


(在上面的UML图中，箭头从派生类指向基类。正如你将会看到的，可以有多个派生类。)

类型不仅仅描述一组对象上的约束，而且还描述与其他类型之间的关系。两个类型可以有共同的特性和行为，但是一个类型可以包括比另一个类型更多的特性，也可以处理更多的消息（或对消息进行不同的处理）。继承表示了基类型和派生类型之间的这种相似性。一个基类型具有所有由它派生出来的类型所共有的特性和行为。程序员创建一个基类型以描述关于系统中的一些对象的思想核心。由这个基类型，我们可以派生出其他类型来表述实现该核心的不同途径。

例如，垃圾再生机器要对垃圾进行分类。这里基类型是“垃圾”，每件垃圾有重量、价值等，并且可以被破碎、被融化或被分解。这样，可以派生出更特殊的垃圾类型，它们可以有另外的特性（瓶子有颜色）或行为（铝可以被压碎，钢可以带有磁性）。另外，有些行为可以不同（纸的价值取决于它的种类和情况）。使用继承，我们可以建立类型的层次结构，在该层次结构中用其类型术语来表述我们需要解决的问题。

第二个例子是经典的“形体”范例，可以用于计算机辅助设计系统或游戏模拟。在这里，基类型是“形体”，每个形体有大小、颜色、位置等。每个形体能被绘制、擦除、移动、着色等。由此，可以派生出特殊类型的形体：圆形、正方形、三角形等，它们中的每一个都有另外的特性和行为。例如，某些形体可以翻转。有些行为可以不同，形体面积的计算。类型层次结构既体现了形体之间的相似性，又体现了它们之间的区别。

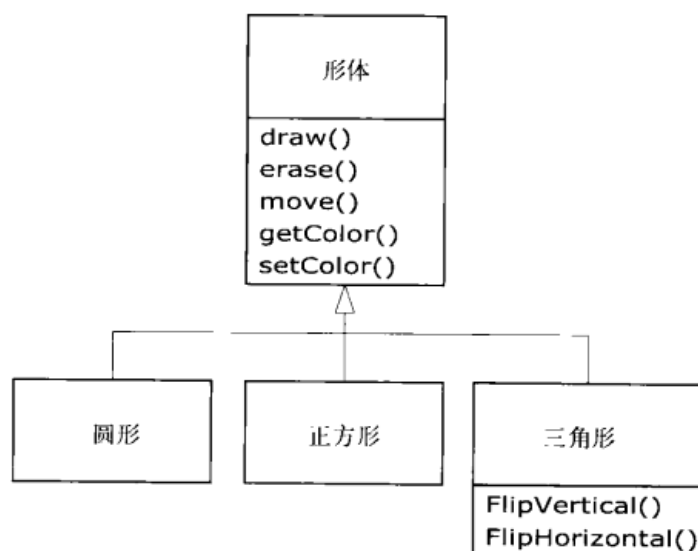


用与问题相同的术语描述问题的解是非常有益的，因为这样我们就无须在问题的描述和解的描述之间使用许多的中间模型。使用对象，类的层次结构就是最初的模型，所以能直接从实际世界中的系统描述进入代码中的系统描述。实际上，使用面向对象设计，人们的困难之一是从开始到结束过于简单。一个已经习惯于寻找复杂解的训练有素的头脑，往往会被问题本来的简单性所难住。

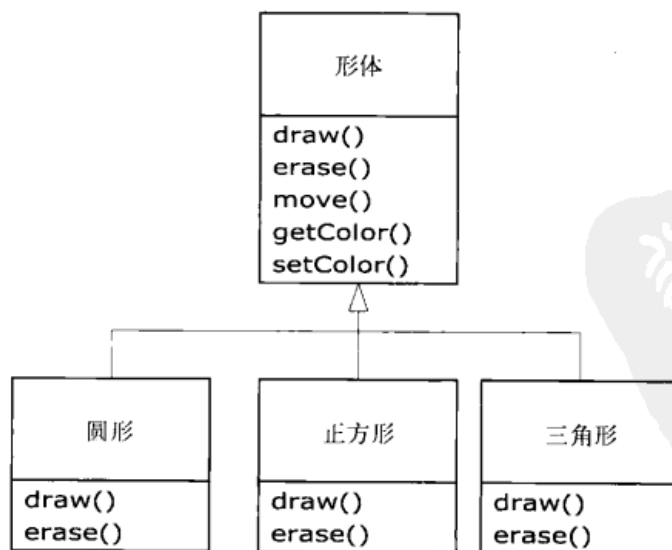
当我们从已经存在的类型来继承时，我们就创造了一个新类型。这个新类型不仅包含那个已经存在的类型的所有成员（虽然私有成员已被隐藏且不可访问），但更重要的是，它复制了这个基类的接口。也就是说，所有能够发送给这个基类对象的消息，也能够发送给这个派生类的对象。因为我们能够根据发送给一个类的消息知道这个类的类型，所以这意味着这个派生类与这个基类是相同类型的。在前面的例子中，“圆形是一个形体”。这种通过继承实现类型等价性，是理解面向对象程序设计含义的基本途径之一。

由于基类和派生类有相同接口，因此伴随着接口必然有一些实现。也就是说，当对象接收到一个特定的消息后必定执行一些代码。如果只是简单地继承一个类，而不做其他任何事情，来自基类接口的方法也就直接进入了派生类。这就意味着，派生类的对象不仅有相同的类型，而且有相同的行为，这一点并不是特别有意义的。

有两种方法能使新派生类区别于原始基类。第一种相当直接，简单地向派生类添加全新的函数。这些新函数不是基类接口的一部分。这意味着，这个基类不能做我们希望它做的事情，所以必须添加函数。继承的这种简单和原始的运用有时就是问题的完美解。但是，我们会进一步看到，基类可能也需要这些添加的函数。在面向对象程序设计中，这种设计的发现和迭代过程会经常发生。



虽然继承有时意味着向接口添加新函数，但这未必真的需要。使新类有别于基类的第二个和更重要的方法是，改变已经存在的基类函数的行为，这称为重载（*overriding*）这个函数。

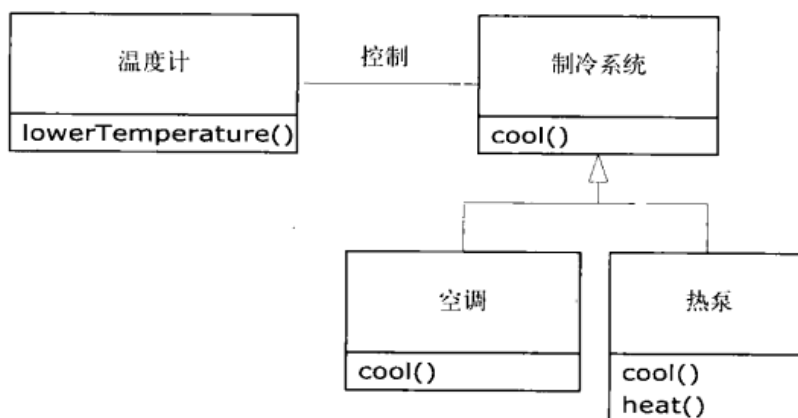


要覆盖函数，可以直接在派生类中创建新定义。相当于说：“我正在使用同一个接口函数，但是我希望它为我的新类型做不同的事情。”

### 1.5.1 is-a关系和is-like-a关系

对于继承有一些争论。继承应当只覆盖基类（并且不添加基类中没有的新成员函数）吗？这就意味着派生类与基类是完全相同的类型，因为它们有相同的接口。结果是，我们可以用派生类的对象代替基类的对象。这被认为是纯代替（*pure substitution*），常常被称做代替原则（*substitution principle*）。在某种意义上，这是对待继承的理想方法。我们常把基类和派生类之间的关系看做是一个“is-a（是）”关系，因为我们可以说“圆形是一个形体”。对继承的一种测试方法就是看我们是否可以说这些类有“is-a”关系，而且还有意义。

有时需要向一个派生类型添加新的接口元素，这样就扩展了接口并创建了新类型。这个新类型仍然可以代替这个基类，但这个代替不是完美的，因为这些新函数不能从基类访问。这可以描述为“is-like-a（像）”关系；新类型有老类型的接口，但还包含其他函数，所以不能说它们完全相同。以一台空调为例。假设你的房子与制冷的全部控制连线；也就是说，它有一个允许你控制冷却的接口。设想这台空调坏了，用一台热泵代替它，这台热泵既可以制冷又可以制热，这台热泵就像一台空调，但它能做更多的事情。因为你的房子的控制系统仅仅是针对制冷功能设计的，所以它仅限于与新对象的制冷部分通信。新对象的接口已经被扩展，而这个已经存在的系统只知道原来的接口，并不知道扩展的部分。



很显然，基类“制冷系统”是不充分的，应当改为“温度控制系统”，使它也能包含加热功能。在这一点上，代替原则可用。上图是一个例子，它既可以发生在设计过程中，也可以发生在现实世界中。

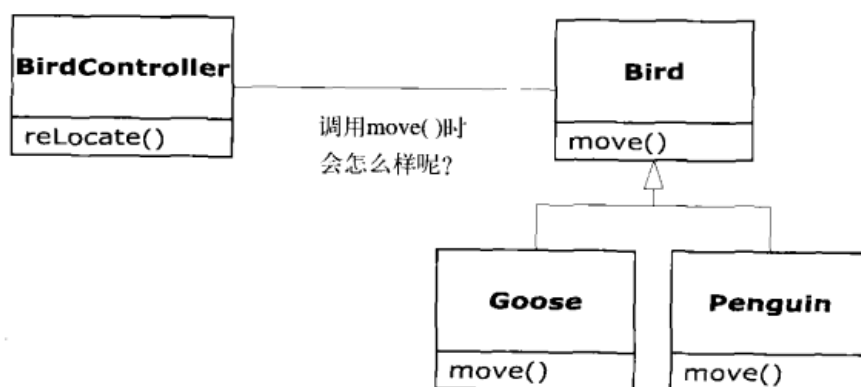
当我们考虑代替原则，很容易将这种方法（纯代替）看做是做事的惟一方法。实际上，如果我们的设计能够采用这种方法，效果也很好。但是，我们还发现有时必须向派生类的接口添加新函数。通过考察，我们发现两种情况都很常见。

## 1.6 具有多态性的可互换对象

当处理类型层次结构时，程序员常常希望不把对象看做是某一特殊类型的成员，而是想把它看做是其基本类型的成员，这样就允许程序员编写不依赖于特殊类型的程序代码。在形体的例子中，函数可以对一般形体进行操作，而不关心它们是圆形、正方形还是三角形。所有的形体都能被绘制、擦除和移动，所以这些函数能简单地发送消息给一个形体对象，而不考虑这个对象如何处理这个消息。

这样，程序代码不受增添新类型的影响，而且增添新类型是扩展面向对象程序来处理新情况最普通的方法。例如，可以派生出形体的一个新的子类型，称为五边形，而不必修改那些处理一般形体的函数。通过派生新的子类型，可以很容易扩展程序，这个能力很重要，因为这会在降低软件维护费用的同时，极大地改善软件设计。

然而，如果试图把派生类型的对象看做是它们所属的基本类型（圆形看做形体，自行车看做车辆，鸕鹚看做鸟），这里就有一个问题：如果一个函数告诉一个一般的形体去绘制它自己，或者告诉一个一般的车辆去行驶，或者告诉一只一般的鸟去飞翔，则编译器在编译时就不能确切地知道应当执行哪段代码。同样的问题是，消息发送时，程序员并不想知道将执行哪段代码。绘图函数能等同地应用于圆形、正方形或三角形，对象根据它的特殊类型来执行合适的代码。如果增加一个新的子类型，不用修改函数调用，它就可以执行不同的代码。编译器不能确切地知道执行哪段代码，那么它应该怎么办呢？例如，在下图中，**BirdController**对象只是与一般的**Bird**对象交互，并不知道它们到底是什么类型。这对于**BirdController**是方便的，因为不需要编写专门的代码来确定它正在对哪种**Bird**工作以及它有什么样的行为。但当忽略专门的**Bird**类型而调用**move()**时，将发生什么事情呢？会出现正确的行为吗？（**Goose**是跑、是飞、还是游泳？**Penguin**是跑、还是游泳？）



在面向对象的程序设计中，答案是非常新奇的：编译器并不做传统意义上的函数调用。由非OOP编译器产生的函数调用会导致与被调用代码的早捆绑（*early binding*），对于这一术语，读者可能还没有听说过，因为从来没有想到过它。早捆绑的意思是，编译器会对特定的函数名产生调用，而连接器将这个调用解析为要执行代码的绝对地址。在OOP中，直到程序运行时，编译器才能确定执行代码的地址，所以，当消息被发送给一般对象时，需要采用其他的方案。

为了解决这一问题，面向对象语言采用晚捆绑（*late binding*）的思想。当给对象发送消息时，在程序运行时才去确定被调用的代码。编译器保证这个被调用的函数存在，并执行参数和返回值的类型检查〔其中不采用这种处理方式的语言称为弱类型（*weakly typed*）语言〕，但是它并不知道将执行的确切代码。

为了执行晚捆绑，C++编译器在真正调用的地方插入一段特殊的代码。通过使用存放在对象自身中的信息，这段代码在运行时计算被调用函数函数体的地址（这一过程将在第15章中详细介绍）。这样，每个对象就能根据这段二进制代码的内容有不同的行为。当一个对象接收到消息时，它根据这个消息判断应当做什么。

我们可以用关键字**virtual**声明他希望某个函数有晚捆绑的灵活性。我们并不需要懂得**virtual**使用的机制，但是没有它，我们就不能用C++进行面向对象的程序设计。在C++中，

必须记住添加**virtual**关键字，因为根据规定，默认情况下成员函数不能动态捆绑。**virtual**函数（虚函数）可用来表示出在相同家族中的类具有不同的行为。这些不同是产生多态行为的原因。

考虑形体的例子，在本章的前面画出过类的家族（所有基于统一接口的类）。为了论述多态性，我们希望编写一段简单的代码，只涉及基类，不涉及类型的具体细节。这段代码是从特定类型信息中分离出来的，编写起来比较简单，理解起来也比较容易。如果有一个新的类型（例如**Hexagon**）通过继承添加进来，那么所写的这段代码将会适用于“**Shape**”的这个新类型，就像在已经存在的类型上使用一样。这样，程序就是可扩展的（*extensible*）。

如果用C++编写一个函数（很快，读者将会学习如何去写）：

```
void doStuff(Shape& s) {
    s.erase();
    // ...
    s.draw();
}
```

这个函数与任何**Shape**对话，所以它独立于它正在绘制或擦除的对象的具体类型（‘&’表示“取这个对象的地址，传给**doStuff()**”，但是现在理解这些细节并不重要）。如果在程序的其他部分使用**doStuff()**函数：

```
Circle c;
Triangle t;
Line l;
doStuff(c);
doStuff(t);
doStuff(l);
```

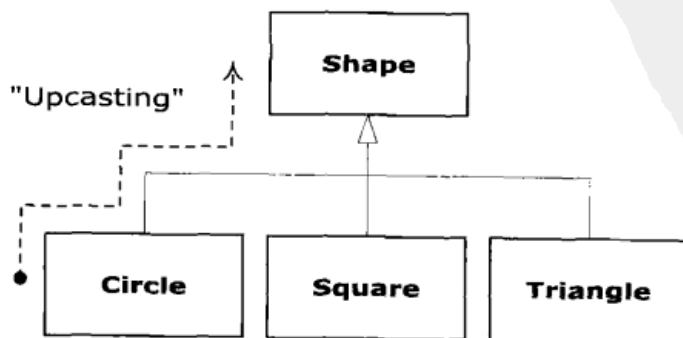
对**doStuff()**的调用会自动正确工作，而不管调用对象的确切类型。

这真是一个令人惊讶的技术。想一想这行代码：

```
doStuff(c);
```

这里发生的事情是将**Circle**传递给对**Shape**有效的函数。因为**Circle**是一个**Shape**，所以可以由**doStuff()**处理。这就是说，任何能由**doStuff()**发送给**Shape**的消息，**Circle**都能接收。所以它是完全安全的和符合逻辑的。

我们把处理派生类型就如同处理其基类型的过程称为向上类型转换（*upcasting*）。“cast”一词来自铸造领域，“up”一词来自于继承图的典型排列方式，基类型置于顶层，派生类向下层展开。这样，类型向基类型的转换是沿继承图向上移动，即“向上类型转换”。





面向对象程序在一些地方会包含一些向上类型转换，因为这正是我们从必须了解所处理的是什么具体类型这一桎梏中解脱出来的方式。请看在doStuff()中的代码：

```
s.erase();
// ...
s.draw();
```

注意，这可不是在说：“如果是Circle，做这件事，如果是Square，做这件事，等等”。如果写这种代码，要检查Shape所有可能的类型，那将是很糟糕的，因为每次添加一种新的Shape，都必须改变代码。这里，我们只是在说：“它是一种形体，我知道它能erase()和draw()自己，如法操作，注意细节的正确性。”

doStuff()代码最引人注意的地方是它能做正确的事情。对Circle调用draw()将执行的代码不同于对Square或Line调用draw()所执行的代码。但是当draw()的消息发送给一个匿名Shape时，正确行为的发生取决于这个Shape的实际类型。这是令人惊奇的，因为，如前所述，当C++编译器编译doStuff()代码时，它并不知道它所处理对象的准确类型。所以以此类推，似乎最终调用的是Shape的erase()和draw()的版本，而不是特殊的Circle、Square或Line的版本。然而，因为多态性，一切操作都完全正确。编译器和运行系统可以处理这些细节，我们只需要知道它会这样做和知道如何用它设计程序就行了。如果一个成员函数是virtual的，则当我们给一个对象发送消息时，这个对象将做正确的事情，即便是在有向上类型转换的情况下。

## 1.7 创建和销毁对象

从技术角度，OOP的论域就是抽象数据类型、继承和多态性。但是，其他一些问题也是重要的。本节对这些问题给出综述。

特别重要的是对象创建和销毁的方法。对象的数据存放在何处？如何控制对象的生命期？不同的程序设计语言有不同的行事之道。C++采取的方法是把效率控制作为最重要的问题，所以它为程序员提供了一个选择。为了最大化运行速度，通过将对象存放在栈中或静态存储区域中，存储和生命期可以在编写程序时确定。栈是内存中的一个区域，可以直接由微处理器在程序执行期间存放数据。在栈中的变量有时称为自动变量（*automatic variable*）或局部变量（*scoped variable*）。静态存储区域简单说是内存的一个固定块，在程序开始执行以前分配。使用栈或静态存储区，可以快速分配和释放，有时这是有价值的。然而，我们牺牲了灵活性，因为程序员必须在写程序时知道对象的准确数量、生命期和类型。如果程序员正在解决一个更一般的问题，例如计算机辅助设计、仓库管理或者空中交通控制，这就太受限制了。

第二种方法是在称为堆（*heap*）的区域动态创建对象。用这种方法，可以直到运行时还不知道需要多少个对象，它们的生命期是什么和它们的准确的数据类型是什么。这些决定是在程序运行之中作出的。如果需要新的对象，直接使用new关键字让它在堆上生成。当使用结束时，用关键字delete释放。

因为这种存储是在运行时动态管理的，所以在堆上分配存储所需要的时间比在栈上创建存储的时间长得多（在栈上创建存储常常只是一条向下移动栈指针的微处理器指令，另外一条是移回指令）。动态方法做出了一般性的逻辑假设，即对象趋向于更加复杂，所以，为找出存储和释放这个存储的额外开销对于对象的创建没有重要的影响。另外，对于解决一般性的程序设计问题，最大的灵活性是主要的。

另一个问题是对象的生命期。如果在栈上或在静态存储上创建一个对象，编译器决定这个对象持续多长时间并能自动销毁它。然而，如果在堆上创建它，编译器则不知道它的生命期。在C++中，程序员必须编程决定何时销毁此对象。然后使用`delete`关键字执行这个销毁任务。作为一个替换，运行环境可以提供一个称为垃圾收集器 (*garbage collector*) 的功能，当一个对象不再被使用时此功能可以自动发现并销毁这个对象。当然，使用垃圾收集器编写程序是非常方便的，但是它需要所有应用软件能忍受垃圾收集器的存在及垃圾收集的系统开销。这并不符合C++语言的设计需要，因此C++没有包括它，尽管存在用于C++的第三方垃圾收集器。

## 1.8 异常处理：应对错误

从程序设计语言出现开始，错误处理就是最难的问题之一。因为设计一个好的错误处理方案非常困难，许多语言忽略这个问题，将这个问题转交给库的设计者，而库的设计者往往采取不彻底的措施，即可以在许多情况下起作用，但很容易被绕开，通常是被忽略。大多数错误处理方案的一个主要问题，在于一相情愿地认为程序员会小心地遵循一些语言本身并不强制要求而是商定好的规范。如果程序员不够小心（这种情况在他们非常匆忙的时候经常发生），这些方案就会很容易被忘记。

异常处理 (*exception handling*) 将错误处理直接与程序设计语言甚至有时是操作系统联系起来。异常是一个对象，它在出错的地方抛出，并且被一段用以处理特定类型错误的、相应的异常处理代码 (*exception handler*) 所捕获。异常处理似乎是另一个并行的执行路径，在出错的时候被调用。由于它使用一个单独的执行路径，它不需要干涉正常的执行代码。因为不需经常检查错误，代码可以很简洁。另外，一个抛出的异常并不同于一个由函数返回的错误值或为了指出错误条件而由函数设置的标记，后两者可以被忽略，而异常不能被忽略，必须保证它们在某些点上进行处理。最后，异常提供了一个从错误状态中进行可靠恢复的方法。除了从这个程序中退出以外，我们常常还可以作出正确的设置，并且恢复程序执行，这有助于产生更健壮的系统。

异常处理并不是面向对象的一个特性，尽管在面向对象语言中异常通常用一个对象表示。异常处理的出现早于面向对象语言。

异常处理在本卷中介绍得很少且很少使用；第2卷详尽讨论了异常处理。

## 1.9 分析和设计

面向对象范式是一种新的、不同的编程思考方式，许多人一开始在学习如何处理一个OOP项目时都会感到非常困难。但是了解到任何事物都被认为是对象，并且学会用面向对象的风格去进一步思考之后，我们就可以开始利用OOP所提供的所有优点创造出“好的”设计。

方法 (*method*) [通常称为方法论 (*methodology*)] 是一系列的过程和探索，用以降低程序设计问题的复杂性。自从面向对象程序设计出现，已有许多OOP方法被提出来。本节将让读者感受使用一种方法时应完成什么任务？

尤其在OOP中，方法论是一个充满实验的领域，因此在我们考虑采用一个方法之前，理解它试图要解决什么是重要的。这在C++中尤其正确，这种编程语言在表达一个程序时试图减少复杂性（同C相比）。这在实际可能减缓对更复杂方法论的需求。相反，简单的方法可以满足在C++中处理更大类的问题，在过程型语言中用简单方法处理的问题相比起来则小很多。

认识到术语“方法论”通常太大且承诺太多也是很重要的。设计和编写一个程序时，我们所做的一切就是一个方法。它可能是我们自创的方法，我们可能没有意识到正在创造一种方法，但它确实是我们创造时经历的一个过程。如果它是一个有效的过程，只需要略加调整以和C++配合。如果我们对效率的程序生产方式不满意，就可以考虑采纳一个正式的方法或在许多正式方法中选择某些部分。

经历开发过程时，最重要的问题是：不要迷路。这很容易做到。大部分分析和设计方法都是为了解决最大的一些问题。记住，大多数项目并不适合这一点，因此我们通常可以用一个相对小的子集成功地进行分析和设计<sup>①</sup>。但是采用某种过程，不论它怎么有局限，总比一上来就直接编码好得多。

在开发过程中很容易受阻，陷入“分析瘫痪状态”，这种状态中往往由于没有弄清当前阶段的所有小细节而感到不能继续了。记住，不论做了多少分析，总有系统的一些问题直到设计时才暴露出来，并且更多的问题是到编程或直到程序完成运行时才出现。因此，迅速进行分析和设计并对提出的系统执行测试是相当重要的。

这个问题值得强调。因为我们在过程型语言上的历史经验，一个项目组希望在进入设计和实现之前认真处理和理解每个细节，这是值得赞扬的。的确，在构造DBMS时，需要彻底理解用户的需要。但是DBMS属于能很好表述和充分理解的一类问题。在许多这种程序中，数据库结构就主要是问题之所在。本章讨论的编程问题属于所谓“不定(wild card)”(本人的术语)类型，这种问题的解决方法不是将众所周知的解决方案简单地重组，而是包含一个或多个“不定要素”——先前没有较了解的解决方案的要素，为此，需要研究<sup>②</sup>。由于在分析阶段没有充分的信息去解决这类问题，因此在设计和执行之前试图彻底地分析“不定型”问题会造成分析瘫痪。解决“不定型”问题需要在整个循环中反复，且需要冒风险(这是很有意义的，由于是在试图完成一些新颖的且潜在回报很高的事情)。看起来似乎有风险是由于“匆忙”进入初步实现而引起的，但这样反而能降低风险，因为我们正在较早地确定一个特定的方法对这个问题是不是可行的。产品开发也是一种风险管理。

经常有人提到“建立一个然后丢掉”。在OOP中，我们仍可以将一部分丢掉，然而由于代码被封装成类，在第一次迭代中我们将必然生成一些有用的类设计，并且产生一些不必抛弃的关于系统设计的有价值的思想。因此，在问题的第一次快速遍历中不仅要为下一遍分析、设计及实现产生关键的信息，而且为下一遍建立代码基础。

也就是说，如果我们正在考虑的是一个包含丰富细节而且需要许多步骤和文档的方法学，将很难判断什么时候停止。应当牢记我们正在努力寻找的是什么：

- (1) 有哪些对象？(如何将项目分成多个组成部分？)
- (2) 它们的接口是什么？(需要向每个对象发送什么信息？)

只要知道了对象和接口，就可以编写程序了。由于各种原因我们可能需要比这些更多的描述和文档，但是我们需要信息不能比这些更少。

整个过程可以分5个阶段完成，阶段0只是使用一些结构的初始约定。

① 一个极好的例子是Martin Fowler所写的专著《UML Distilled》(Addison-Wesley, 2000)，该书将复杂的UML过程简化为可管理的子集。

② 我估计这样的项目有一条经验规则：如果不确定因素不止一个，在没有创建一个能工作的原型之前，不要计划它将用多长时间和将花费多少。这里的自由度太大了。

### 1.9.1 第0阶段：制定计划

我们必须首先决定在此过程中应当有哪些步骤。这听起来简单（事实上，所有听起来都挺简单的），但是人们常常在开始编码之前没有考虑这一问题。如果计划是“让我们一开始就编码”，那很好（有时，当我们对问题充分理解时，这是合适的）。至少，我们承认这是一个计划。

在这个阶段，我们可能还要决定一些另外的过程结构，但不是全部。可以理解，有些程序员喜欢用“休假方式”工作，也就是在开展他们的工作过程中，没有强制性的结构。“想做的时候就做”。这可能在短时间内是吸引人的，但是我发现，在进程中设立一些里程碑可以帮助集中我们的注意力，激发我们的热情，而不是只注意“完成项目”这个单一的目标。另外，里程碑将项目分成更细的阶段使得风险减小（此外里程碑还提供更多庆祝的机会）。

当我开始研究小说结构时（有一天我也要写小说），我最初是反对结构思想的，我觉得自己在写作时，直接下笔千言就行了。但是，稍后我认识到，在写涉及计算机的文字时，本身结构足够清晰，所以不需要多想。但是我仍然要组织文字结构，虽然在我头脑中是半意识的。即便我们认为自己的计划只是一上来就开始编码，在后续阶段仍然需要不断询问和回答一些问题。

#### 1.9.1.1 任务陈述

无论建造什么系统，不管如何复杂，都有其基本的目的，有其要处理的业务，有所满足的基本需要。通过依次审视用户界面、硬件或系统的特殊细节、算法编码和效率问题，我们将最终找出它的核心，通常简单而又直接。就像来自好莱坞电影的所谓高层概念（*high concept*），我们能说一句或两句话表述。这种纯粹的表述是起点。

高层概念相当重要，因为它设定了项目的基调，这是一种任务陈述。我们不必一开始就让它正确（我们也许正处于在项目变得完全清晰之前的最后阶段），但是要不停地努力直至它越来越正确。例如：在一个空中交通指挥系统中，我们可以从关于正在建立的系统的一个高层概念入手：“塔楼程序跟踪飞机”。但是当我们将这一系统收缩以适用于一个非常小的机场时，考虑将发生什么情况；可能只有一个控制人员甚至什么都没有。一个更有用的模型不当像它描述问题那样多地关注正在创建的解决方案，例如“飞机到达、卸货、维修、重新装货和离开等”。

### 1.9.2 第1阶段：我们在做什么

在上一代程序设计[称为过程型设计（*procedural design*）]中，这一阶段称为“建立需求分析（*requirements analysis*）和系统规范说明（*system specification*）”。这些当然是容易迷路的地方。它们是一些名字很吓人的文档，本身可能就是大项目。当然，它们的目的是好的。需求分析说的是“制定一系列指导方针，我们将利用它了解任务什么时候完成且用户什么时候满足”。系统规范说明指出，“这是程序将做什么（不是如何做）以满足需求的一个描述”。需求分析实际上是我们和用户之间的一个合同（即使用户在我们的公司工作或是另一些对象或系统）。系统规范说明是对问题的一个顶层探测，且在一定程度上要说明项目是否能做和将需多少时间。由于这两个问题需要人们的共识（并且因为他们通常会随时间的推移而改变意见），我认为最好将它们尽可能地保持最小限度（理想情况下，是列表和基本的图表）以节省时间。我们可能有其他的限制，如需要将它们扩展为大一些的文档，但是小而简洁的最初文档，可以在由一个动态创建描述的领导者的带领下，通过很少几次头脑风暴（*brainstorming*）讨

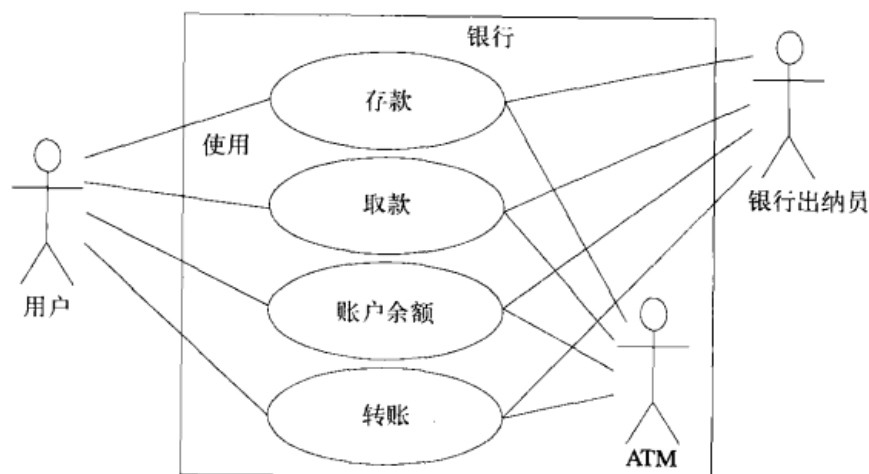
论而得到。这不仅需要每个人的投入，而且能激励小组中每个成员参与，使他们意见一致。也许，最重要的是，它可以使项目以极大的热情开始。

这一阶段中我们有必要把注意力始终放在核心问题上：确定这个系统要做什么。为此，最有价值的工具是一组所谓的“用例 (use case)”。用例指明了系统中的关键特性，它们将展现我们使用的一些基本的类。它们实际上是对类似下述这些问题的描述性回答<sup>①</sup>：

- 1) “谁将使用这个系统？”
- 2) “执行者用这个系统做什么？”
- 3) “执行者如何用这个系统工作？”
- 4) “如果其他人也做这件事，或者同一个执行者有不同的目标，该怎么办？（揭示变化）”
- 5) “当使用这个系统时，会发生什么问题？（揭示异常）”

例如，如果设计一个自动取款机，此系统的一个特定功能方面的用例能够描述这台自动取款机在任何可能情况下的行为。这些“情况”每一个称为情节 (scenario)，而用例可以认为是情节的集合。我们可以把情节认为是以“如果……系统将怎样？”开头的问题。例如，“如果一个用户在24小时内刚刚存了一张支票，且在此支票之外该账户中没有足够的钱能满足提款要求，这时自动取款机怎么办？”。

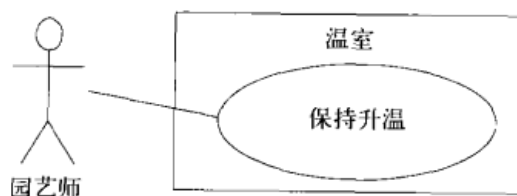
下面的用例图特意进行了简化，以防止我们过早地陷入到系统的实现细节问题中去。



每个小人代表一个“执行者 (actor)”，它通常是一个人或其他类型的自由代理（甚至可以是其他计算机系统，如“ATM”中的情况）。方框代表系统的边界。椭圆代表用例，是对此系统能完成的有价值工作的描述。在执行者和用例之间的直线代表交互。

只要符合用户的使用感受，系统实际上如何实现并不重要。

用例不必十分复杂，即便底层系统非常复杂。这只是为了表示用户眼中的系统形象。例如：



<sup>①</sup> 感谢James H Jarrett的帮助。



通过确定用户在系统中可能有的所有交互行为，用例就生成了需求规范说明。我们试图找到系统的完整用例，完成之后，我们就得到了系统任务的核心内容。注意力集中在用例上的好处是，它们总是将我们带回到要点部分而不至于留心那些对完成任务无关紧要的问题。也就是说，如果得到了全部用例就可以描述系统并进入到下一个阶段。在最初的尝试中我们很难完全得到特征，但这已经很好了。任何事物随着时间的推移都会自己暴露出来，如果在这一点上就要求系统规范说明完美将会永远止步不前。

如果遇到了困难，我们可以通过使用一个近似工具启动这个阶段：用很少的段落描述此系统，然后寻找名词和动词。名词往往意味着执行者、用例的上下文（如“休息室”）或在用例中使用的物品。动词往往意味着执行者和用例之间的交互行为，表示用例中的特定步骤。我们还将发现名词和动词在设计阶段将对应产生对象和消息（注意用例描述子系统间的交互，因此“名词和动词”技术只能作为集体讨论工具，因为它不生成用例）<sup>①</sup>。

用例和执行者之间的边界能指出用户界面的存在，但不能定义这样的用户界面。为了定义和创建用户界面，可参看Larry Constantine和Lucy Lockwood所著的《Software for Use》(Addison Wesley Longman, 1999) 或访问[www.ForUse.com](http://www.ForUse.com)。

虽然这有点像魔术，但此时进行某种基本的进度安排是重要的。我们现在有了创建目标的总体概念，所以我们可能产生它需要多长时间的概念。这里涉及大量因素。如果估算了一个长时间表，则公司可能决定不创建它（并把资源用在更合理的项目上去，这是好事）。或者管理人员可能已经决定这个项目应当花多少时间，并且试图改变我们做出的估计。但是最好一开始有一个准确的时间表，解决早期决心的问题。已经有大量的努力以产生精确建立时间表的技术（就像预测股票市场的技术），然而，最好的方法或许是依靠我们的经验和直觉。得到实际上将花多少时间的估计，加倍，再加上百分之十。我们的直觉也许是对的，我们能及时得到能用的产品。“加倍”将使产品更好，加百分之十用于最后的润色和细化<sup>②</sup>。然而，我们需要解释它，克服抱怨和当我们拿出这个时间表时发生种种事情，最终完成这一问题。

### 1.9.3 第2阶段：我们将如何建立对象

在这一阶段，我们必须做出设计，描述这些类和它们如何交互。确定类和交互的出色技术是类职责协同（*Class-Responsibility-Collaboration, CRC*）卡片。此技术的部分价值是它非常简单：只要有一组3到5英寸的空白卡片，在上面书写。每张卡片描述一个类，在卡片上写的内容是：

(1) 类的名字。这很重要，因为名字体现了类行为的本质，所以有一目了然的作用。

(2) 类的职责：它应当做什么。通常，它可以仅由成员函数的名字陈述（因为在好的设计中，这些名字应当是描述性的），但并不产生其他的注记。如果需要开始这个过程，请从一个懒程序员的立场看这个问题：你希望有什么样的对象魔术般地出现，把你的问题全部解决？

(3) 类的协同：它与其他类有哪些交互？“交互”是非常宽泛的术语。它可以是一些已经

① 更多有关用例的内容可以在Schneider & Winters所写的专著《*Applying Use Cases*》(Addison-Wesley, 1998) 和Rosenberg所写的专著《*Use Case Driven Object Modeling with UML*》(Addison-Wesley, 1999)中找到。

② 我个人观点后来已经变了。加倍和增加百分之十将给出相当准确的估计（假设这里没有太多的不定要素），但是我们仍然需要勤奋工作，以及时完成。如果我们希望时间真的花这么长，并且在这个过程中得到乐趣，我认为，正确的增加是3到4倍。

存在的其他对象对这个类的对象提供的服务。协同还应当考虑这个类的观众。例如如果创建了**Firecracker**（鞭炮），那么谁将观察它，是**Chemist**（药剂师）还是**Spectator**（观众）？前者希望知道鞭炮由什么化学成分组成，后者对鞭炮爆炸后的颜色和形状有反应。

我们可能想让卡片更大一些，因为我们希望从中得到全部信息，但是它们是非常小的，这不仅能保持我们的类小，而且能防止过早地陷入过多的细节。如果一张小卡片上放不下类所需要的信息，那么这个类就太复杂了（或者是考虑过细了，或者应当创建多个类）。理想的类应当一目了然。CRC卡片的思想是帮助我们找到设计的第一印象，使得我们能得到总体概念，然后精炼我们的设计。

CRC卡片的最大的好处之一是在交流中。在一个组中，最好实时进行交流，而不是用计算机。每个人负责几个类（起初它们没有名字或其他信息）。每次只解决一个情节，决定发送什么消息给不同的对象以满足每个情节，这样就能作出一个比较形象的对问题的模拟。当我们经历了这个过程后，就会找出我们所需要的类以及它们的职责和协同，这样，我们同时也填写好这些卡片。当我们完成所有用例后，就有了一个相当完整的设计的第一印象。

在我开始用CRC卡片之前，当提出最初的设计时，我最成功的咨询经验就是站在一个没有OOP经验的项目组前，在白板上描述对象。我们讨论对象应当如何互相通信，擦除其中的一些，用其他的对象替换它们。实际上我是在白板上管理所有的“CRC卡片”。项目组（他们知道项目的目标）真正地在做这个设计，他们“拥有”这个设计，而不是获得既成的设计。我所做的所有事情就是通过提问正确的问题，提炼这些假设，并且从项目组得到反馈，修改这些假设来指导这个过程。这个过程真正好处是项目组学习了如何做面向对象的设计，不是通过复审抽象的例子，而是通过在一个设计上工作，这对于他们是最有兴趣的。

制作了一组CRC卡片之后，我们可能希望用UML<sup>①</sup>创建这个设计的更形式化的描述。我们并不是非要用UML，但它可能有帮助，特别是如果我们要将一个图表挂在墙上，让大家一起思考时，这是一个很好的想法。除了UML之外的另一选择是对象及其接口的文字描述，这或许依赖于我们的程序设计语言，也就是代码本身<sup>②</sup>。

UML还提供了另外一种图形符号来描述系统的动态模型。在一个系统或子系统的状态转换占主导地位，以至于它们需要自己的图表的情况下，这是有帮助的（例如在控制系统中）。我们可能还需要描述数据结构，因为系统或子系统中数据结构是重要因素（例如数据库）。

当已经描述了对象及其接口后，第2阶段就要完成了。这时已经知道了对象中的大多数，通常会有对象漏掉，直到第3阶段才被发现。这没问题。我们关心的是最终能找到所有的对象。在这个阶段较早地发现它们是好的。因为OOP提供了充分的结构，所以如果我们稍迟发现它们也可以。事实上，对象设计可能在程序设计全过程的五个阶段中都会发生。

#### 1.9.3.1 对象设计的五个阶段

对象的设计生命期不仅仅限于写程序的时间。实际上，它出现在一系列阶段上。接受这种观点很有好处，因为我们不再期望设计立刻尽善尽美，而是认识到，对对象做什么和它应当像什么的理解，会随着时间的推移而呈现。这个观点也适用于不同类型程序的设计。特殊类型程序的模式是通过一次又一次地求解问题而形成的（设计模式在第2卷介绍）。同样，对象有自己的模式，通过理解、使用和重用而形成。

① 对于初学者，我推荐上面提到过的专著《UML Distilled》。

② Python (www.Python.org)常常被用做“可执行伪代码”。

(1) **对象发现** 这个阶段出现在程序的最初分析期间。对象可以通过寻找外部因素及边界、系统中重复的元素和最小概念单元而发现。如果已经有了一组类库，某些对象是很明显的。类之间的共同性（暗示着基类和继承关系），可以立刻出现或在设计过程的后期出现。

(2) **对象装配** 当我们正在建立对象时会发现需要一些新成员，这些新成员在对象发现时期未出现过。对象的这种内部需要可能要用新类去支持它。

(3) **系统构造** 再次指出，对对象的更多要求可能出现在以后阶段。随着不断学习，我们会改进我们的对象。与系统中其他对象通信和互相连接的需要，可以改变已有的类或要求新类。例如，我们可以发现需要辅助类，这些类如像一个链表，它们包含很少的状态信息或没有状态信息，只有帮助其他类的功能。

(4) **系统扩充** 当我们向系统增添新的性能时，可能发现我们先前的设计不容易支持系统扩充。这时，我们可以重新构造部分系统，并很可能要增加新类或类层次。

(5) **对象重用** 这是对类真正的强度测试。如果某些人试图在全新的情况下重用它们，他们也许会发现一些缺点。当我们修改一个类以适应更新的程序时，类的一般原则将变得更清楚，直到我们有了一个真正可重用的对象。然而，不要期望从一个系统设计而来的大多数对象是可重用的，大量对象是对于特定系统的。可重用类一般共性较少，为了重用，它们必须解决更一般的问题。

#### 1.9.3.2 对象开发准则

下述步骤提出了考虑开发类时要用到的一些准则：

- 1) 让特定问题生成一个类，然后在解决其他问题期间让这个类生长和成熟。
- 2) 记住，发现所需要的类（和它们的接口），是设计系统的主要内容。如果已经有了那些类，这个项目就不困难了。
- 3) 不要强迫自己在一开始就知道每一件事情，应当不断地学习。
- 4) 开始编程，让一些部分能够运行，这样就可以证明或否定已生成的设计。不要害怕过程型大杂烩式的代码——类的隔离性可以控制它们。坏的类不会破坏好的类。
- 5) 尽量保持简单。具有明显用途的不太清楚的对象比很复杂的接口好。当需要下决心时，用Occam的Razor方法：选择简单的类，因为简单的类总是好一些。从小的和简单的类开始，当我们对它有了较好的理解时再扩展这个类接口，但是很难从一个类中删去元素。

#### 1.9.4 第3阶段：创建核心

这是从粗线条设计向编译和执行可执行代码体的最初转换阶段，特别是，它将证明或否定我们的体系结构。这不是一遍的过程，而是反复地建立系统的一系列步骤的开始，我们将在第4阶段中看到这一点。

我们的目标是寻找实现系统体系结构的核心，尽管这个系统在第一遍不太完整。我们正在创建一个框架，在将来的反复中可以完善它。我们正在完成第一遍多系统集成和测试，向客户（stakeholder）提出反馈意见，关于他们的系统看上去如何以及如何发展等。理想情况下，我们还可以暴露一些严重的问题。我们大概还可以发现对最初的体系结构能做哪些改变和改进——本来在没有实现这个系统之前，可能是无法了解这些内容的。

建立这个系统的一部分工作是实际检查，就是对照需求分析和系统规范说明与测试结果

(无论需求分析和规范说明以何种形式存在)。确保我们的测试结果与需求和用例符合。当系统核心稳定后,我们就可以向下进行和增加更多的功能了。

#### 1.9.5 第4阶段:迭代用例

一旦代码框架运行起来,我们增加的每一组特征本身就是一个小项目。在一次迭代(iteration)期间,我们增加一组特征,一次迭代是一个相当短的开发时期。

一次迭代有多长时间?理想情况下,每次迭代为一到三个星期(具体随实现语言而异)。在这个期间的最后,我们得到一个集成的、测试过的、比前一周期有更多功能的系统。特别有趣的是迭代的基础:一个用例。每个用例是一组相关功能,在一次迭代中加入系统。这不仅为我们更好地提供了“用例应当处于什么范围内”的概念,而且还对用例概念进行了巩固,在分析和设计之后这个概念并未丢弃,它是整个软件建造过程中开发的基本单元。

当我们达到目标功能或外部最终期限到了,并且客户对当前版本满意时,我们就停止迭代。(记住,软件行业是建立在双方约定的基础之上。)因为这个过程是迭代的,所以我们有许多机会交货,而不是只有一个终点;开放源代码项目是在一次迭代的和高反馈的环境中开发,而这正是它成功的原因。

有许多理由说明迭代开发过程是有价值的。我们可以更早地揭露和解决严重问题,客户有足够的机会改变它们的意见,程序员会更满意,能更精确地掌握项目。而另一个重要的好处是对风险承担者(stakeholder)意见的反馈,他们能从项目当前状态准确地看到各方面因素。这可以减少或消除令人头脑昏昏然的会议,增强风险承担者的信心和支持。

#### 1.9.6 第5阶段:进化

这是开发周期中,传统上称为“维护”的一个阶段,是一个含义广泛的术语,包含了从“让软件真正按最初提出的方式运行”到“添加用户忘记说明的性能”,到更传统的“排除暴露的错误”和“在出现新的需求时添加性能”。所以,对术语“维护”有许多误解,它已经有点虚假的成分,部分因为它假设我们已经实际上建立了原始的程序,且所有的需要就是改变其中一些部分,加加油,防止生锈。也许,有更好的术语来描述所进行的工作。

此处将使用术语“进化”(evolution)<sup>①</sup>。这就是说,“我们不可能第一次就使软件正确,所以应该为学习、返工和修改留有余地”。当我们对问题有了深入的学习和领会之后,可能需要做大量的修改。如果我们使软件不断进化,直到使软件正确,无论在短期内还是在长期内,将产生极为优雅的程序。进化是使程序从好到优秀,是使第一遍不理解的问题变清楚的过程。它也是我们的类能从只为一个项目使用进化为可重用资源的过程。

“使软件正确”的意思不只是使程序按照要求和用例工作,还意味着我们理解代码的内部结构,并且认识到它能很好地协同工作,没有拙笨的语法和过大的对象,也没有难看的暴露的代码。另外,必须认识到,程序结构将经历各种修改而保全下来,这些修改贯穿整个生命期,也要认识到,这些修改可以是很容易进行和很简洁的。这可不是小成就。我们不仅必须懂得我们正在建造的程序,而且必须懂得这个程序将进化[我称之为改变矢量(vector of change)<sup>②</sup>]。

① 进化的一个方面在Martin Fowler的专著《Refactoring: improving the design of existing code》(Addison-Wesley, 1999)中做了介绍。需预先警告,这本书的例子全部使用JAVA编写。

② 这一术语在第2卷的“设计模式”一章中研究。

幸运的是，面向对象程序设计语言特别适合支持这样连续的修改，由对象创建的边界是防止结构被破坏的保障。面向对象程序设计语言还允许我们做大幅度改变而不引起代码的全面动荡，这在过程型程序中似乎太剧烈了。事实上，支持进化可能是OOP的最重要的好处。

借助于进化，我们创建了近似于我们所要创建的程序，然后进行检查，与需求比较，看哪些地方有缺点。随后回头看，重新设计和重新实现程序不能正确工作的部分，改进它<sup>①</sup>。在得到正确解决方案之前，可能实际上需要几次解决这个问题或问题的一个方面。（对设计模式的研究在第2卷中描述，对此阶段非常有用。）

进化还发生在我们建造系统时，开始它好像符合需求，以后又发现它实际上不是想要的系统。当看到这个系统运行时，我们发现实际上想要解决的是另一个问题。如果我们认为这种进化将会发生，则应该尽快地建造第一个版本，这样可以发现它实际上是不是想要的系统。

也许，需要记住的最重要的事情是，按默认情况（实际上是按定义），如果修改了一个类，则它的超类和子类都仍然正常工作。不要害怕修改（特别是，如果我们已经有内部的一组单元测试，验证了修改的正确性时）。修改不一定会破坏程序，结果的任何改变都将限定于子类和/或被改变的类的协同者。

### 1.9.7 计划的回报

当然了，谁也不会没有仔细计划之前，就建造房子。然而，如果建造鸡圈或狗圈，计划就不需要那么精细了，但是可能仍然以某种草图开始，指导建造过程。软件开发已经走向了极端。在很长的时间里，人们在开发中没有多少结构概念，很多大项目都失败了。其结果最终使各种方法学应运而生，它们包含大量的结构和细节，首先主要针对大项目。这些方法学大得可怕，并不好用，看上去好像我们要花所有的时间在写文档，没有时间编写程序。（真的常常如此。）我希望在这里提出的是走中间道路因地制宜。用一种能满足需要（和个性化）的方法。无论选择的方法学多么小，在项目中进行一些计划还是会有较大改进的，比完全没有计划强得多。请记住，根据各种估计，超过50%的项目都失败了（有些估计说70%以上）。

遵循计划（简单和短小的更加适宜），在编码之前就提出设计结构，我们会发现，事情总的说来比一上来就编码的方法容易得多，并且会认识到大多数情况是令人满意的。就我的经验，提出一个漂亮的方案实际上是在一种完全不同水平上的满足，感觉上更接近于艺术，而不是技术。精致总是有回报的，这不是一种虚浮的追求。它不仅给出了一个容易建造和调试的程序，而且容易理解和维护，这就是其经济价值的体现。

## 1.10 极限编程

我从研究生时开始就断断续续研究过分析和设计技术。极限编程（*eXtreme Programming*, XP）的思想在我见过的方法学中最为激进也最令人愉快。在由Kent Beck编写的《*Extreme Programming Explained*》（Addison-Wesley, 2000）<sup>②</sup>一书和在Web站点[www.xprogramming.com](http://www.xprogramming.com)上可以找到它的历史。

XP既是程序设计工作的哲学，又是做程序设计的一组原则。其中一些原则反映在新近出

① 这是有些像“快速生成原型”，先建造一个快而稍差（*quick-and-dirty*）的版本，然后研究这个系统，再丢弃这个原型并正确地建造它。快速生成原型的麻烦是人们不丢弃这个原型，而是在它上面建造。与过程型程序设计中缺乏结构相结合，这常常会产生混乱的系统，使得维护费用提高。

② 本书的中文版已由人民邮电出版社出版。——编辑注



现的其他方法学中，但我认为，有两个原则最重要、贡献最大，即“先写测试”和“结对编程”。虽然Beck强烈坚持全过程，但他也指出，如果只采用这两项实践，就能极大地改进生产效率和可靠性。

### 1.10.1 先写测试

测试传统上被归于项目的最后阶段，在“万事俱备，只欠肯定”之后。这意味着它的优先级很低，专门做此工作的人没有足够的地位，甚至可怜兮兮地只能在地下室里干活，不算“真正的程序员”。测试组对此的反应是，穿着黑色的衣服，当攻破了某个程序时就兴高采烈（老实说，当我攻破了C++编译器时我也有这种感觉）。

XP革命性地改变了测试的这个概念，将它置于与编码相等（甚至更高）的优先地位。事实上，我们需要在编写被测试代码之前写测试，而且这些测试与代码永远在一起。这些测试必须在每次项目集成时都能成功地执行（这是经常地，有时一天几次）。

先写测试有两个极其重要的作用。

第一，它强制类的接口有清楚的定义。我经常建议，人们设计系统时会把解决特定的问题的理想的类，作为工具。XP的测试策略比这走得更远，它准确地指明这个类看上去必须像什么，对于这个类的用户，这个类准确地如何动作。用确切的术语，可以写所有的文档描述，或者创建所有图表，描述一个类应当如何行动和看上去像什么，但是什么都比不上一组测试真实。前者列出的是期望，而测试是由编译器和运行程序强制的合约。很难想象有比测试更具体的类描述。

当创建测试时，我们被迫要充分思考这个类，常常会发现所需要的功能，而这些功能可能会在思考UML图、CRC、用例等过程期间漏掉。

写测试的第二个重要的作用，是能在每次编译软件时运行这些测试。这实际上让编译器又执行了测试。如果从这个角度观察程序设计语言的发展，就会发现在技术上的改进实际上是围绕着测试开展的。汇编语言只检查语法，而C增加了一些语义约束，能防止程序员犯某些类型的错误。OOP语言强加了更多的语义约束，可以把它们看成是某种实际形式的测试。“这个数据类型的用法合适吗？这个函数的调用合适吗？”这些都是由编译器或运行时系统执行的测试任务。我们已经看到了在程序设计语言中加入这些测试的成效：人们能用更少的时间和劳动写更复杂的程序，并使它们运行。开始我不理解为什么能有如此成效，后来认识到，这正是测试的作用。如果程序员写错了，内部测试的安全网就告诉他有问题，并指出在什么地方。

但是，由语言设计提供的内部测试只能做部分的工作。有时，我们必须自己动手，增加其余的测试，形成配套（与编译器和运行时系统协作），验证程序的一切。正如编译器能陪伴帮助我们一样，我们也希望其他的测试也能从一开始就帮助我们。这就是为什么我们要书写测试、并在每次系统创建时自动地运行它们的原因。我们的测试就变成了这个语言提供的安全网的扩充。

在功能越来越强大的程序设计语言中，我发现可以更大胆地尝试更多易错的实验，因为我知道高级语言功能会帮助排除错误，避免浪费时间。XP测试机制对于整个项目的作用也是如此。因为我们知道测试始终能捕捉我们引进的任何问题（当需要时我们有规律地增加新的测试），所以当需要时可以做大的改变，不用担心会使整个系统混乱。这真是太强大了。

### 1.10.2 结对编程

结对编程 (pair programming) 反对深植于我们心中的个人主义, 我们从小就通过学校 (在那里, 成功与失败全在自己, 与邻座一起工作这样的事情会被认为是“欺骗”) 和媒体 (特别是好莱坞电影, 其中的英雄总是在与盲目服从作斗争) 在灌输这种思想<sup>①</sup>。程序员也被认为是个人主义的典范, 正如Larry Constantine喜欢说的“牛仔编码者”。而XP, 这一打破传统的方法学, 主张代码应当在每个工作站上由两个人编写。而且这应当在有一堆工作站的工作场合中进行, 拆掉人们喜欢的隔板。实际上, Beck说, 转向XP的第一个任务是用螺丝刀和螺钉完成的, 拆除挡道的一切东西<sup>②</sup> (这需要经理能说服后勤部门)。

结对编程的好处是, 一个人编写代码时另一个人在思考。思考者的头脑中保持总体概念, 不仅是手头问题的这一段, 而且还有XP指导方针。例如, 如果两个人都在工作, 就不太可能会有其中的一个说“我不想首先写测试”而愤然离去。如果编码者遇到障碍, 他们就交换位置。如果两个人都遇到障碍, 他们的讨论可能被在这个区域工作的其他人听到, 可能给出帮助。这种结对方式, 使事情顺畅、有章可循。也许更重要的是, 它能使程序设计更具有社交性和娱乐性。

我在一些讲习班的练习期间用过结对编程, 似乎明显地改进了每个人的练习过程。

## 1.11 为什么C++会成功

C++能够如此成功, 部分原因是它的目标不只是为了将C语言转变成OOP语言 (虽然这是最初的目的), 而且还为了解决当今程序员, 特别是那些在C语言中已经大量投入的程序员所面临的许多问题。传统上, 人们已经对OOP语言有了这样的看法: 程序员应当抛弃所知道的每件事情并且从一组新概念和新文法重新开始, 程序员应当相信, 从长远观点来看, 最好是丢掉所有来自过程型语言的老行装。从长远角度看, 这是对的, 但从短期角度看, 这些行装还是有价值的。最有价值的可能不是那些原有的代码库 (用合适的工具, 可以转变它), 而是原有的头脑库。作为一个职业C程序员, 如果让他丢掉他知道的关于C的每一件事情, 以适应新的语言, 那么, 在几个月内, 他将毫无成果, 直到他的头脑适应了这一新范例时为止。但如果他能调整已有的C知识, 并在这个基础上扩展, 那么他就可以继续保持高的生产效率, 带着已有的知识, 进入面向对象程序设计的世界。因为每一个人都有自己的程序设计思维模型, 所以这个转变是很混乱的。因此, 简而言之, C++成功的原因是很经济的: 转变到OOP上需要代价, 而转变到C++上所花的代价可能比较小<sup>③</sup>。

C++的目的是提高生产效率。生产效率与多方面因素有关, 而语言是为了尽可能地帮助使用者, 尽可能少地因为使用武断的规则或特殊性能的需求而妨碍使用者。C++成功的原因是它立足于实际: 尽可能地为程序员提供最大利益 (至少从C的观点上看是这样)。

### 1.11.1 一个较好的C

即便程序员在C++环境下继续写C代码, 也能直接得到好处, 因为C++堵塞了C语言中的

① 虽然这个观点可能太美国化了, 但是好莱坞的故事遍布世界。

② (尤其是) 包括PA系统。我一度在某公司工作, 他们坚持广播每个管理人员的电话, 这种做法经常会打断我们的工作 (但是管理者都不能想到把它去掉)。最后, 当没人注意时, 我就剪断了电线。

③ 我说“可能”, 因为C++太复杂了, 实际上转变到Java上可能更便宜。决定选择哪种语言有许多因素, 本书中我假定已经选择了C++。

许多漏洞，并提供更好的类型检查和编译时的分析。程序员必须先声明函数，使编译器能检查它们的使用。预处理器也限制了值替换和宏，这就减少了查找错误的困难。C++有一个特征，称为引用 (*reference*)，它允许对函数参数和返回值的地址进行更方便的处理。通过函数重载 (*function overloading*)，改进了对名字的处理，使程序员能对不同的函数使用相同的名字。另外，一个称为名字空间 (*namespaces*) 的特征也改进了对名字的控制。除此之外，还有许多较小的特征改善了C的安全性。

### 1.11.2 延续式的学习过程

与学习新语言有关的问题是生产效率问题。所有公司都很难承受因为软件工程师正在学习新语言而突然降低了生产效率。C++是对C的扩充，而不是新的文法和新的程序设计模型。当程序员学习和理解这些性能时，他可以逐渐应用它们，这就允许他继续创建有用的代码。这是C++成功的最重要的原因之一。

另外，已有的C代码在C++中仍然是有用的，但因为C++编译器是更严格的，所以，重新编译这些代码时，常常会发现隐藏的错误。

### 1.11.3 效率

有时，以牺牲程序执行速度换取程序员的生产效率是值得的。假如，一个金融模型仅在短期内有用，那么，快速创建这个模型比所写程序能更快速执行更重要。然而，很多应用程序都要求一定程度的运行效率，所以C++在更高运行效率方面总是有些偏差。但因为C程序员通常具有很强的效率意识，所以这也保证他们并不认为这个语言太庞大、太慢。C++的一些性能允许程序员在产生的代码不够有效时做一些改善。

C++不仅有与C相同的低层控制能力（和在C++程序中直接写汇编语言的能力），而且非正式的证据表明，面向对象的C++程序的速度与用C写的程序的速度相差在 $\pm 10\%$ 之内，而且常常更接近<sup>①</sup>。用OOP方法设计的程序实际上可能比C的对应版本更有效。

### 1.11.4 系统更容易表达和理解

为适合于某问题而设计的类当然能更好地表达这个问题。这意味着编写代码时，程序员是在用问题空间的术语描述问题的解（例如“把垫圈放进材料箱”），而不是用计算机的术语，也就是解空间的术语，来描述问题的解（例如“设置芯片的一位，即合上继电器”）。程序员所涉及的是较高层的概念，单行代码能做更多的事情。

易于表达所带来的另一个优点是易于维护。据报道，在程序的整个生命周期中，维护占了花费的很大一部分。如果程序容易理解，那么它就更容易维护，这还能减少创建和维护文档的花费。

### 1.11.5 尽量使用库

创建程序最快的方法是使用已经写好的代码：库。C++的主要目标是让程序员能更容易地使用库，这是通过将库转换为新数据类型（类）来完成的。引入一个库，就是向该语言增加

---

<sup>①</sup> 参看Dan Saks在“C/C++ User's Journal”杂志上的栏目，对C++库性能的重要调查。

一个新类型。因为是由编译器负责这个库如何使用，也就是保证适当的初始化和清除，保证函数正确调用，所以程序员的精力可以集中在他想要这个库做什么，而不是如何做这件事。

因为名字能够在程序的各部分之间隔离，所以程序员想使用多少库就使用多少库，不会有像C语言那样的名字冲突。

#### 1.11.6 利用模板的源代码重用

有一些重要的类型的类，它们要求修改源代码以有效地重用它们。C++的模板 (*template*) 功能可以自动完成对代码的修改，因而它是重用库代码的特别有用的工具。用模板设计的类型很容易与其他类型一起工作。因为模板对客户程序员隐藏了这类代码重用的复杂性，所以它很有好处。

#### 1.11.7 错误处理

在C语言中，错误处理是一个很糟糕的问题。程序员常常会忽视它们，而且常常对它们束手无策。如果正在建立庞大而复杂的程序，没有什么比让错误隐藏在某处，且没有引导告诉我们它来自何处更糟的了。C++的异常处理 (*exception handling*) (在本卷介绍，在第2卷中将有完整的讨论，其材料可以从[www.BruceEckel.com](http://www.BruceEckel.com)上下载) 可以保证能检查到错误并使特定的某件事情发生。

#### 1.11.8 大型程序设计

许多传统语言对程序的规模和复杂性有内在的限制。例如，BASIC对于某些类型的问题能很快解决，但是如果这个程序有几页纸长，或者超出该语言的正常解题范围，那么它可能永远也算不出结果。C语言同样有这样的限制，例如当程序超过50 000行时，名字冲突就开始成为问题。实际上，程序员会用光函数和变量名。另一个特别糟糕的问题是如果C语言中存在一些小漏洞——有错误隐藏在大程序中，要找出它们是极其困难的。

并没有清楚的文字告诉程序员，什么时候他的语言会失效，即便有，他也会忽视它们。他不说“我的BASIC程序太大，我必须用C重写”，而经常是试图硬塞进另外几行，以增加额外的功能。所以额外的花费就悄悄加上来了。

设计C++的目的是为了辅助大型程序设计，这就是说，去掉这些在小程序和大程序之间的复杂性的分界。当程序员写hello-world之类实用程序时，他确实不需要用OOP、模板、名字空间和异常处理，但是当他需要的时候，这些性能就有用了。而且，编译器在排除错误方面，对于小程序和大程序一样有效。

### 1.12 为向OOP转变而采取的策略

如果决定采用OOP，我们的下一个问题可能是“如何才能使得经理/同事/部门/伙伴开始使用OOP？”想想看，作为独立的程序员，应当如何学习使用新语言和新的程序设计形式。和前面一样，首先训练和做例子，再通过一个试验项目得到一个基本的感觉，不要做太混乱的任何事情，然后尝试做一个“真实世界”的实际有用的项目。在第一个项目中，通过读、向专家问问题、与朋友切磋等方式，继续我们的训练。基本上，这就是许多有经验的程序员建议的从C转到C++的方法。转变整个公司当然应当采用某些动态的方法，但回忆个人是如何

做这件事的，能在转变的每一步中起帮助作用。

### 1.12.1 指导方针

当向OOP和C++转变时，有一些方针要考虑：

#### 1.12.1.1 训练

第一步是某种形式的培训。记住公司在原始C代码上的投资，并且当每个人都在为这些遗留的东西而为难时，应努力在6到9个月内不使公司完全陷入混乱。挑选一个小组进行培训，更适宜的情况是，这个小组成员是一些勤奋好学、能很好地在一起工作的人们，当他们正在学习C++时，能形成他们自己的支持网。

有时建议采用另一种方法，即对公司各级人员同时进行培训，包括为策略经理而开设的概论课程，以及为项目开发者而开设的设计课程和编程课程。对于较小的公司或较大公司的下层，对他们做事情的方法做一些基本的改变是非常好的。因为代价较高，所以一些公司可能选择以项目层训练而开始，做导航式的项目（可能请一个外面的导师），然后让这个项目组变成公司其他人的老师。

#### 1.12.1.2 低风险项目

首先尝试一个低风险项目，并允许出错。一旦得到了一些经验，就将这第一个小组的成员安插进其他项目组中，或者用这个组的成员作为OOP的技术顶梁柱。这第一个项目可能不能正确工作，所以该项目不应是公司的关键任务。它应当是简单的、自成一体和有指导意义的。这意味着它应当包括创建对于公司的其他程序员学习C++有意义的类。

#### 1.12.1.3 来自成功的模型

在动手之前，挑一些好的面向对象设计的例子。很可能有些人已经解决过我们的问题，如果他们还没有正确地解决它，我们可以应用已经学到的关于抽象的知识，来修改存在的设计，以适合我们自己的需要。这是设计模式的一般概念，将在第2卷中详细讲解。

#### 1.12.1.4 使用已有的类库

转变为C++的主要经济动机是容易使用以类库形式存在的代码（特别是标准C++库，将在本书的第2卷中深入探讨），最短的应用开发周期是利用现有库创建和使用对象，除了main()以外不必自己写任何东西。然而，一些新程序员并不理解这一点，不知道已有的类库，或出于对语言的迷恋希望写可能已经存在的类。如果在转变过程的早期努力查找和重用其他人的代码，那么我们在OOP和C++方面将得到最好的成功。

#### 1.12.1.5 不要用C++重写已有的代码

虽然用C++编译C代码通常会有（有时是很大的）好处，它能发现老代码中的问题，但是把时间花在对已有的功能代码用C++重写上，通常不是时间的最佳利用方法（如果必须翻译成对象，我们可以用C++类“包装”C代码）。特别是，如果代码是为重用而编写的，会有很大的好处。但是，有可能出现这种情况：在最初的几个项目中，并不能看到生产效率如您梦想的一样增长，除非这是新项目。如果是从概念到实现的项目，C++和OOP表现最为出色。

### 1.12.2 管理的障碍

如果我们是经理，我们的工作是为项目组争取资源，搬除通往胜利道路上的障碍，并且



通常要努力提供更高的生产效率和和谐的环境,使项目组更有可能产生奇迹。转向C++的三类方式,如果不花费任何代价都是不可能的。与C程序员(也可能是其他过程型语言的程序员)项目组的OOP替代品相比,虽然转向C++可能更便宜一些(这取决于约束条件)<sup>①</sup>,但并不是免费的,在试图说服公司转向C++并对转移投资之前,我们应当知道会有障碍。

#### 1.12.2.1 启动的代价

转移到C++的代价比获得C++编译器(最好的编辑器之一,GNU C++编译器,是免费的)大得多。进行培训(也可能是第一个项目的指导),并且确定和购买解决问题的类库而不是自己开发,那么中长期的代价就将减小。这种直接的经费开支是实际建议所必须考虑的因素。另外还有隐藏的花费,在于学习新语言和新程序设计环境期间的生产效率下降。培训和指导确实能减少花费,但是项目组成员必须自己克服了解新技术的各种困难。在这个过程中,将犯更多错误(这是特点,因为认识错误是学习的最快途径),生产效率下降。尽管如此,对于一些类型的程序设计问题,有了正确的类和正确的开发环境,C++学习时可能会比继续用C语言有更高的生产效率(即便考虑到程序员正在犯更多的错误和每天写更少的代码行)。

#### 1.12.2.2 性能问题

一个普遍的问题是,“OOP不会自动使得我们的程序变大和变慢吗?”回答是“不一定”。大多数传统的OOP语言是以实验和快速原型方法设计的,这样实际上就决定了其在规模上的扩大和在速度上的下降。然而,C++是以生产性程序的方式设计的。当用快速原型方式时,我们能尽可能快地将构件组合在一起,而忽视效率问题。如果使用了第三方库,通常已经由它们的厂商优化过了,在这种情况下,用快速开发方法,效率也不是问题。如果我们有一个喜欢的系统,它足够小和快,就继续使用,如果不是,就调整,用描述工具(*profiling tool*),首先改进速度。这可用简单的C++内部功能完成。如果无效,就寻找对底层实现的修改,但要做到不改变所需要的特殊类。只有当全都不能解决问题时,才需要改变设计。性能在设计中的地位很重要,是主要的设计标准之一。运用快速原型法,可以尽早地了解系统性能。

如前所述,在C和C++之间的规模和速度之比常常不同,但一般是10%之内,而且通常更接近。当使用C++代替C时,可能在规模和速度上得到大的改进,因为C++所做的设计很大程度上不同于C所做的。

在C和C++之间比较规模和速度的证据至今还只是传说性的估计,也许还会继续如此。尽管有一些人建议对相同的项目用C和C++同时做,但也许不会有公司把钱浪费在这里,除非它非常大并且对这个研究项目感兴趣。即便如此,它也希望钱花得更好。已经从C(或其他过程型语言)转到C++(或一些其他OOP语言)的程序员几乎一致地都有在程序设计效率上得到很大提高的个人经验,这是能找到的最引人注目的证据。

#### 1.12.2.3 常见的设计错误

当项目组开始使用OOP和C++时,程序员们将会出现一系列常见的设计错误。这经常会发生,因为在早期项目的设计和实现过程中从专家们那里得到的反馈太少,在公司中没有专家,而聘请顾问可能有阻力。我们可能会觉得,在这个周期中,我们懂得OOP太早了并开始了一条不好的道路。有时,对于在这个语言上有经验的一些人而言,显而易见的问题可能是新手们在内部的激烈争论。大量的这类问题都能通过聘用外部富有经验的专家培训和指导来避免。

<sup>①</sup> 因为生产效率的改进,所以Java语言也应当被考虑。

另一方面，容易出现设计错误的事实也反映出C++的主要缺点：对C向后兼容（当然，这也是它的主要优势）。为了完成能编译C代码的任务，C++不得不做一些妥协，这形成了一些“死角”。这些都是事实，并且包含了学习这个语言的大量弯路。在本书和后续的卷（以及其他书，参看附录C）中，试图揭示当使用C++时会遇到的大量陷阱。应当知道，在这个安全网中有一些漏洞。

### 1.13 小结

本章希望使读者对面向对象程序设计和C++的大量问题有一定的感性认识，包括为什么OOP是不同的，为什么C++特别不同，什么是OOP方法的思想，和最终当公司转到OOP和C++时会遇到的各种问题。

OOP和C++可能不一定对每个人都适合。对自己的需要作出估计，并决定是否C++能很好地满足自己的需要，或者是否用别的程序设计系统（包括当前正在用的这种系统）会更好，这是很重要的。如果读者知道，在可预见的未来自己的需要非常专门化，如果有特殊的约束，不能由C++满足，那么可以自己研究替代物<sup>①</sup>。即便最终选择了C++，也至少应当懂得这些选择是什么，并应当对为什么取这个方向有清晰的看法。

我们已经知道过程型程序的概貌：数据定义和函数调用。为了理解这样程序的含义，必浏览函数调用和底层概念，在头脑中创建一个模型。这就是我们设计过程型程序时需要中间描述的原因，这些程序往往是混乱的，因为表达的术语更偏向于计算机而不是所解决的问题。

因为C++对C语言增加了许多新概念，所以我们自然会认为在C++中的`main()`会比等价的C程序复杂得多。在这里，我们将很高兴地看到：一个写得好的C++程序一般比等价的C程序简单得多和更容易理解。我们将看到的是描述问题空间概念的对象定义（而不是计算机描述的问题）和发送给这些对象的消息，这些消息描述了问题空间的活动。面向对象程序设计的乐趣之一是，对于设计良好的程序，通过读程序可以很容易理解它。通常，这里代码更少，因为我们的许多问题都能通过重用库代码解决。

---

<sup>①</sup> 我特别推荐Java(<http://java.sun.com>)和Python (<http://www.Python.org>)。



## 对象的创建与使用

本章介绍一些C++语法和程序构造概念，使读者能编写和运行一些简单的面向对象的程序。下一章，我们再详细介绍C和C++的基本语法。

首先通过学习本章，我们会对C++面向对象编程的风格有个基本的了解，进而明白人们热衷于这种语言的一些原因。这些足以引导读者读完第3章的内容。第3章中包含了大量的C语言细节，几乎是对C语言的详尽无遗的介绍。

用户定义的数据类型或类（*class*），是C++区别于传统过程型语言的地方。类是一种新的数据类型，用来解决特定问题。一旦创建了一个类，任何人都可以使用它而不需知道它的构造方式和工作原理细节。本章仅把类作为C++内置的另一种数据类型来使用。

通常将创建好的类存放在库里。本章会使用几个C++的类库。一个很重要的标准库是输入输出流库，可以用它从文件或键盘读取数据，并且将数据写入文件和显示出来。我们还将看到来自标准C++类库中的非常方便的**string**类和**vector**容器。到本章结束时，我们会发现使用预先定义好的类库是很容易的事情。

为了创建我们的第一个程序，我们必须先了解用于创建应用程序的工具。

### 2.1 语言的翻译过程

任何一种计算机语言都要从某种人们容易理解的形式（源代码）转化成计算机能执行的形式（机器指令）。通常，翻译器分为两类：解释器（*interpreter*）和编译器（*compiler*）。

#### 2.1.1 解释器

解释器将源代码转化成一些动作（它可由多组机器指令组成）并立即执行这些动作。例如，BASIC就是一个流行的解释性语言。传统的BASIC解释器一次翻译和执行一行，然后将这一行的解释丢掉。因为解释器必须重新翻译任何重复的代码，程序执行就变慢了。为了提高速度，也可对BASIC进行编译。现在许多的解释器，诸如Python语言的解释器，先把整个程序转化成某种中间语言，然后由执行速度更快的解释器来执行<sup>⊖</sup>。

使用解释器有许多好处。从写代码到执行代码的转换几乎能立即完成，并且源代码总是现存的，所以一旦出现错误，解释器能很容易地指出。对于解释器，较好的交互性和适于快速程序开发（不必要求可执行程序）也是常被提到的两个优点。

当做大项目时解释性语言有某些局限性（而Python似乎是一个例外）。解释器（或其简化版）必须驻留内存以执行程序，而这样一来，即使是最快的解释器其速度也会变得让人难以接受。大部分的解释器要求一次输入整个源代码。这不仅造成内存空间的限制，而且如果语言不提供设施来隔离不同代码段之间的影响，一旦出现错误，就很难调试。

⊖ 解释器和编译器之间的界限非常模糊，尤其对Python来说，它具有编译语言的许多特点和功能，但它只是一种解释语言的快速转换。

### 2.1.2 编译器

编译器直接把源代码转化成汇编语言或机器指令。最终的结果是一个或多个机器代码的文件。这是一个复杂的过程，通常分几步完成。使用编译器时，从写源代码转到执行代码，是一个较长的过程。

仰仗编译器设计者的聪明才智，编译器生成的程序往往只需较少的运行空间，并且执行速度更快。虽然编译后的程序较小、运行速度快是人们认为应当使用编译器的理由，但在许多时候这却不是最重要的。某些语言（如C语言）可以分别编译各段程序。最后使用连接器（*linker*）把各段程序连接成一个完整的可执行程序。这个过程称为分段编译（*separate compilation*）。

分段编译有许多好处。由于编译器或编译环境的限制，不能一次完成编译的整个程序，可以分段编译。每次创建和测试程序的一部分，当这部分程序能正常运行后，就把它作为程序组块保存起来。人们把测试通过并能正常运行的程序块收集起来加入库（*library*）中，供其他程序员使用。由于独立创建每一段程序，其他各段程序的复杂性便被隐藏起来。所有这些特点支持大型程序的创建<sup>①</sup>。

编译器的调试功能不断地得以改进。早期的编译器只能产生机器代码，要知道程序的运行状态，程序员要插入打印语句。但这样做并不总是有效的。现代编译器能在可执行程序中插入与源代码有关的信息。这个信息由一些强大的源代码层的调试器（*source-level debugger*）使用，以便通过跟踪程序经过源代码的进展来显示程序的执行情况。

为了提高编译速度，一些编译器采用了内存中编译（*in-memory compilation*）。大多数编译器，编译时每一步都要读写文件。内存中编译器就是将编译器程序存放在RAM中。对于小程序来说，内存中编译器几乎能和解释器一样响应。

### 2.1.3 编译过程

为了用C/C++编程，应该了解编译过程的步骤和所需工具。某些语言（特别是C/C++）编译时，首先要对源代码执行预处理。预处理器（*preprocessor*）是一个简单的程序，它用程序员（利用预处理器指令）定义好的模式代替源代码中的模式。预处理器指令用来节省输入，增加代码的可读性。（C++程序设计并不鼓励多使用预处理指令，因为它可能会引起一些不易发现错误，这些将在本书的后面分析）。预处理过的代码通常存放在一个中间文件中。

编译一般分两遍进行。首先，对预处理过的代码进行语法分析。编译器把源代码分解成小的单元并把它们按树形结构组织起来。表达式“**A+B**”中的“**A**”、“**+**”和“**B**”就是语法分析树的叶子节点。

有时候会在编译的第一遍和第二遍之间使用全局优化器（*global optimizer*）来生成更短、更快的代码。

编译的第二遍，由代码生成器（*code generator*）遍历语法分析树，把树的每个节点转化成汇编语言或机器代码。如果代码生成器生成的是汇编语言，那么还必须用汇编器对其汇编。两种情况的最后结果都是生成目标模块（通常是，一个以**.o**或**.obj**为扩展名的文件）。有时也会在第二遍中使用窥孔优化器（*peephole optimizer*）从相邻一段代码中查找冗余汇编语句。

用“object”（目标）一词表示一段机器代码是一种不合适的选择，在面向对象程序设计

<sup>①</sup> Python 又是一个例外，因为它也支持分段编译。

之前这一名词就普遍使用了。在讨论编译时“object”与“goal”（目标）含义相同，而在面向对象程序设计中，它的意思是“一个有边界的事物”。

连接器（*linker*）把一组目标模块连接成为一个可执行程序，操作系统可以装载和运行它。当某个目标模块中的函数要引用另一目标模块中的函数或变量时，由连接器来处理这些引用；这就保证了所有需要的、在编译时存在的外部函数和变量仍然存在。连接器还要添加一个特殊的目标模块来完成程序启动任务。

连接器能搜索称为“库”的特殊文件来处理它的所有引用。库将一组目标模块包含在一个文件中。库由一个被称为库管理器（*librarian*）的程序来创建和维护。

#### 2.1.3.1 静态类型检查

类型检查（*type checking*）是编译器在第一遍中完成的。类型检查是检查函数参数是否正确使用，以防止许多程序设计错误。由于类型检查是在编译阶段而不是程序运行阶段进行的，所以称之为静态类型检查（*static type checking*）。

某些面向对象的语言（如Java）也可在程序运行时作部分类型检查[动态类型检查（*dynamic type checking*）]。动态类型检查和静态类型检查结合使用，比仅仅使用静态类型检查更有效。但它也增加了程序执行的开销。

C++使用静态类型检查，因为C++语言不采用任何特殊的运行时支持来处理错误操作。静态类型检查在编译时就告知程序员类型被误用，从而加快了执行时的速度。通过对C++的学习，我们会看到C++语言的主要设计目标也是追求运行速度快，这与面向生产的编程语言C语言一样。

在C++里可以不使用静态类型检查。我们可以自己做动态类型检查——这只需要写一些代码。

## 2.2 分段编译工具

当创建大的项目时，分段编译尤其重要。在C/C++中，可以将一个大程序构造成为许多小程序块，而这些小程序块容易管理，可独立测试。程序分割的最基本的方法是创建命名子程序。在C和C++里，子程序称为函数（*function*），函数是一段代码段，可以将这些函数放在不同的文件中，并能分别编译。另一种解释，函数是程序的基本单位，因为不能把一个函数分开，让其不同的部分放在不同的文件中；整个函数必须完整地放在一个文件里（尽管文件可拥有不止一个函数）。

当调用函数时，通常要传给它一些参数（*argument*）。这些参数是我们希望函数在执行时使用的值。当函数执行完后，可得到一个返回值（*return value*），返回值是函数作为执行结果返回的一个值。但也可以编写不带参数没有返回值的函数。

程序可由多个文件构成，一个文件中的函数很可能要访问另一些文件中的函数和数据。编译一个文件时，C或C++编译器必须知道在另一些文件中的函数和数据，特别是它的名字和基本用法。编译器就是要确保函数和数据被正确地使用。“告知编译器”外部函数和数据的名称及它们的模样，这一过程就是声明（*declaration*）。一旦声明了一个函数或变量，编译器知道怎样检查对它们的引用，以确保引用正确。

### 2.2.1 声明与定义

声明（*declaration*）和定义（*definition*）这两个术语在整本书中都会准确地使用，因此必须弄清它们之间的区别。事实上，所有的C/C++程序都要求声明。编写第一个程序之前，需要了解声明的基本方法。

声明是向编译器介绍名字——标识符。它告诉编译器“这个函数或这个变量在某处可找到，它的模样像什么”。而定义是说：“在这里建立变量”或“在这里建立函数”。它为名字分配存储空间。无论定义的是函数还是变量，编译器都要为它们在定义点分配存储空间。对于变量，编译器确定变量的大小，然后在内存中开辟空间来保存变量的数据。对于函数，编译器会生成代码，这些代码最终也要占用一定的内存。

在C和C++中，可以在不同的地方声明相同的变量和函数，但只能有一个定义[有时这称为ODR (one-definition rule, 单一定义规则)]。当连接器连接所有的目标模块时，如果发现一个函数或变量有多个定义，连接器将报告出错。

定义也可以是声明。如果定义`int x;`之前，编译器没有发现标识符`x`，编译器则把这一标识符看成是声明并立即为它分配存储空间。

#### 2.2.1.1 函数声明的语法

C/C++的函数声明就是给函数取名、指定函数的参数类型和返回值。例如，下面是一个叫`func1()`的函数声明，它带了两个整数类型的参数（整数类型在C/C++中以关键字`int`表示）并返回一个整数：

```
int func1(int,int);
```

第一个关键字是函数返回值类型：`int`。参数按其使用的顺序依次排在函数后面的括号内。分号说明声明结束，在这种情况下，它告诉编译器“就这些，这里没有函数定义。”

C/C++尽量使声明形式和使用形式一致。例如，假设`a`是另一个整数变量，上面的函数可以如下方式使用：

```
a = func1(2,3);
```

因为`func1()`返回的是一个整数，C/C++编译器要检查`func1()`的使用情况，以确保`a`能接受返回值，并且还要检查函数参数的类型匹配情况。

在函数声明时，可以给参数命名。编译器会忽略这些参数名，但对程序员来说它们可以帮助记忆。例如，我们有下面的形式声明`func1()`，它与前面的声明意义相同：

```
int func1(int length, int width);
```

#### 2.2.1.2 一点说明

对于带空参数表的函数，C和C++有很大的不同。在C语言中，声明

```
int func2();
```

表示“一个可带任意参数（任意数目，任意类型）的函数”。这就妨碍了类型检查。而在C++语言中它就意味着“不带参数的函数”。

#### 2.2.1.3 函数的定义

函数定义看起来像函数声明，但它还有函数体。函数体是一个用大括号括起来的语句集。大括号表示这段代码的开始和结束。为了定义函数体为空的（函数体不含代码）函数`func1()`，应当写为：

```
int func1(int length, int width) { }
```

注意，在函数定义中，大括号代替了分号的作用，因为大括号括起了一条或一组语句，



所以就不需要分号了。另外也要注意，如果要在函数体中使用参数的话，函数定义中的参数必须有名称（上面的函数没有用到定义的参数，因此在这里是可选的）。

#### 2.2.1.4 变量声明的语法

对“变量声明”的解释向来很模糊且自相矛盾，而理解它准确的含义对于正确的理解定义和阅读程序十分重要。变量声明告知编译器变量的外表特征。这好像是对编译器说：“我知道你以前没有看到过这名字，但我保证它一定在某个地方，它是X类型的变量。”

函数声明包括函数类型（即返回值类型）、函数名、参数列表和一个分号。这些信息使得编译器足以认出它是一个函数声明并可识别出这个函数的外部特征。由此推断，变量声明应该是类型标识后面跟一个标识符。例如：

```
int a;
```

可以声明变量**a**是一个整数，这符合上面的逻辑。但这就产生了一个矛盾：这段代码有足够的信息让编译器为整数**a**分配空间，而且编译器也确实给整数**a**分配了空间。要解决这个矛盾，对于C/C++需要一个关键字来说明“这只是一个声明，它的定义在别的地方”。这个关键字就是**extern**，它表示变量是在文件以外定义的，或在文件后面部分才定义。

在变量定义前加**extern**关键字表示声明一个变量但不定义它，例如：

```
extern int a;
```

**extern**也可用于函数声明。例如：

```
extern int func1(int length, int width);
```

这种声明方式和先前的**func1()**声明方式一样。因为没有函数体，编译器必定把它作为声明而不是函数定义。**extern**关键字对函数来说是多余的、可选的。C语言的设计者并不要求函数声明使用**extern**，这可能有些令人遗憾；如果函数声明也要求使用**extern**，那么在形式上与变量声明更加一致，从而减少了混乱（但这就需要更多的输入，这也许能解释为什么不要求函数使用**extern**的原因）。

下面是一些声明的例子：

```
//: C02:Declare.cpp
// Declaration & definition examples
extern int i; // Declaration without definition
extern float f(float); // Function declaration
float b; // Declaration & definition
float f(float a) { // Definition
    return a + 1.0;
}

int i; // Definition
int h(int x) { // Declaration & definition
    return x + 1;
}

int main() {
    b = 1.0;
    i = 2;
    f(b);
    h(i);
} ///:~
```



函数声明时参数标识符是可选的。函数定义时则要求要有标识符（这里指C语言，而C++不要求）。

#### 2.2.1.5 包含头文件

大部分的库包含众多的函数和变量。为了减少工作量，确保一致性，当对这些函数和变量做外部声明时，C/C++使用“头文件”(*header file*)。头文件是一个含有某个库的外部声明函数和变量的文件。它通常是扩展名为“.h”的文件，如headerfile.h（可能还会看到一些较老的程序使用其他扩展名，如“.hxx”或“.hpp”，但现在已经很少了）。

头文件由创建库的程序员提供。为了声明在库中已有的函数和变量，用户只需包含头文件即可。包含头文件，要使用#include预处理器命令。它告诉预处理器打开指定的头文件并在#include语句所在的地方插入头文件。#include有两种方式来指定文件：尖括号(< >)或双引号。

以尖括号指定头文件，如下所示：

```
#include <header>
```

用尖括号来指定文件时，预处理器是以特定的方式来寻找文件，一般是环境中或编译器命令行指定的某种寻找路径。这种设置寻找路径的机制随机器、操作系统、C++实现的不同而不同，要视具体情况而定。

以双引号指定文件，如下所示：

```
#include "local.h"
```

用双引号时，预处理器以“由实现定义的方式”来寻找文件。它通常是从当前目录开始寻找，如果文件没有找到，那么include命令就按与尖括号同样的方式重新开始寻找。

包含iostream头文件，要用如下语句

```
#include <iostream>
```

预处理器会找到iostream头文件（通常在“include”子目录下）并把它插入include语句所在位置。

#### 2.2.1.6 标准C++ include 语句格式

随着C++的不断演化，不同的编译器厂商选用了不同的文件扩展名。而且，不同的操作系统对文件名有不同的限制，特别是对文件名长度限制。结果引起了对源代码的可移植性的限制。为了消除这些差别，标准使用的格式允许文件名长度可以大于众所周知的8个字符，去除了扩展名。例如，代替老式的包含iostream.h的语句

```
#include <iostream.h>
```

现在可以写成：

```
#include <iostream>
```

如果需要截短文件名和加上扩展名，翻译器会按照一定的方式来实现包含语句，以适应特定的编译器和操作系统。当然，如果想使用这种没有扩展名的风格，但编译器厂商没有提供这种支持，也可以将厂商提供的头文件拷贝成没有扩展名的文件。

从C继承下来的带有传统“.h”扩展名的库仍然可用。然而，也可以用更现代的C++风格

使用它们，即在文件名前加一个字母“c”。这样

```
#include <stdio.h>
#include <stdlib.h>
```

就变为:

```
#include <cstdio>
#include <cstdlib>
```

对所有的标准C头文件都一样。这就为读者提供了一个区分标志，说明所使用的是C还是C++库。

新的包含格式和老的效果是不一样的：使用.h的文件是老的、非模板化的版本，而没有.h的文件是新的模板化版本。如果在同一程序中混用这两种形式，会遇到某些问题。

## 2.2.2 连接

连接器把由编译器生成的目标模块（一般是带“.o”或“.obj”扩展名的文件）连接成为操作系统可以加载和执行的程序。它是编译过程的最后阶段。

连接器的特性随系统不同而不同。通常，只需告诉连接器目标模块和要连接的库的名称，及可执行程序的名称，连接器就可以开始执行连接任务了。一些系统要求用户自己调用连接器。很多C++软件包可以让用户通过C++编译器来调用连接器。多数情况下，连接器的调用是不可见的。

某些早期的连接器对目标文件和库文件只查找一次，这些连接器从左到右查找一遍所给的目标文件和库文件列表。因此目标文件和库文件的顺序就特别重要。如果连接的时候遇到一些莫名其妙的问题，就有可能与给定连接器的文件顺序有关。

## 2.2.3 使用库文件

到此，了解了一些基本的术语，现在可以学习如何来使用库了。

使用库必须：

- 1) 包含库的头文件。
- 2) 使用库中的函数和变量。
- 3) 把库连接进可执行程序。

目标模块没有加入库时，也可执行上述步骤。对于C/C++的分段编译，包含头文件和连接目标模块是基本步骤。

### 2.2.3.1 连接器如何查找库

当C或C++要对函数和变量进行外部引用时，根据引用情况，连接器会选择两种处理方法中的一种。如果还未遇到过这个函数或变量的定义，连接器会把它的标识符加到“未解析的引用”列表中。如果连接器遇到过函数或变量定义，那么这就是已解决的引用。

如果连接器在目标模块列表中不能找到函数或变量的定义，它将去查找库。库有某种索引方式，连接器不必到库里查找所有目标模块——而只需浏览索引。当连接器在库中找到定义后，就将整个目标模块而不仅仅是函数定义连接到可执行程序。注意，仅仅是库中包含所需定义的目标模块加入连接，而不是整个库参加连接（否则程序会变得毫无意义的庞大）。如果想尽量减小程序的长度，当构造自己的库时，可以考虑一个源代码文件只放一个函数。这

要求更多的编辑工作<sup>①</sup>，但它对使用者来说是有益的。

因为连接器按指定的顺序查找文件，所以，用户使用与库函数同名的函数，把带有这种函数的文件插到库文件列表之前，就能用他自己的函数取代库函数。由于在找到库文件之前，连接器已先用用户所给定的函数来解释引用，因此被使用的是用户的函数而不是库函数。注意，这可能是一个bug，并且C++名字空间禁止这样做。

### 2.2.3.2 秘密的附加模块

当创建一个C/C++可执行程序时，连接器会秘密连接某些模块。其中之一是启动模块，它包含了对程序的初始化例程。初始化例程是开始执行C/C++程序时必须首先执行一段程序。初始化例程建立堆栈，并初始化程序中的某些变量。

连接器总是从标准库中查找程序中调用的经过编译的“标准”函数。由于标准库总可以被找到，所以只要在程序中包含所需的头文件，就可以使用库中的任何模块，并且不必告诉连接器去找标准库。例如，标准的C++库中有*iostream*函数。要用这些函数，只需包含<iostream>头文件即可。

如果使用附加的库，必须把该库文件名添加到由连接器处理的列表文件中。

### 2.2.3.3 使用简单的C语言库

用C++来编写代码，并不禁止用C的库函数。事实上，整个C的库以默认方式包含在标准的C++库中。这些函数代替用户做了大量的工作，因此，使用它们，可以节约许多时间。

本书将尽可能地使用标准的C++库函数（也包含标准C库函数），但是只有用标准库函数才能保证程序的可移植性。在某些情况下必须使用非标准C++库函数的地方，我们也将尽量使用符合POSIX标准的函数。POSIX是基于UNIX上的一个标准，它包括的函数是C++库中没有的。通常能在UNIX（特别是Linux）平台上找到POSIX函数，也可能在DOS/Windows下找到。例如，如果要用到多线程编程，最好使用POSIX线程库，这样的代码就容易理解、端口通信和维护（POSIX线程库通常只用到操作系统提供的基本的线程设施）。

## 2.3 编写第一个C++程序

现在，已经了解了几乎足够的基础知识，可以创建和编译一个程序了，它将用到标准的C++ *iostream*类。这些*iostream*类可从文件和标准的输入输出设备（通常指控制台，但也可重定向到文件和设备）中读写数据。这个简单的程序将利用流对象在屏幕上显示消息。

### 2.3.1 使用*iostream*类

为了声明*iostream*类中的函数和外部数据，要用如下语句包含头文件：

```
#include <iostream>
```

第一个程序用到了标准输出的概念，标准输出的含义就是“发送输出的通用场所”。在其他例子中会看到使用标准输出的不同方式，但这里指输出到控制台。*iostream*包自动定义一个名为*cout*的变量（对象），它接受所有与标准输出绑定的数据。

<sup>①</sup> 我推荐使用Perl或Python自动完成这项任务作为程序员库打包过程的一部分（参见[www.Pperl.org](http://www.Pperl.org)或[www.Python.org](http://www.Python.org)）。

将数据发送到标准输出，要用操作符“<<”。C程序员知道这个操作符表示“向左移位”，下一章我们将会讨论。应当说向左移位与输出毫无关系。然而，C++允许操作符“重载”。操作符重载后与某种特殊类型的对象一起使用，它就有了新的含义。和*iostream*对象在一起，操作符“<<”意思就是“发送到”。例如

```
cout << "howdy!";
```

意思就是把字符串“howdy!”发送到*cout*对象（*cout*是“控制台输出（console output）”的简写）。

这是操作符重载的初步知识。第12章将详细讨论操作符重载。

### 2.3.2 名字空间

正如第1章所提到的那样，在C语言中，当程序达到一定规模之后，会遇到的一个问题是“用完了”函数名和标识符。当然，并非我们真正用完了所有函数名和标识符，而是简单地想出一个新名称就不太容易了。更重要的是，当程序达到一定的规模之后，通常分成许多块，每一块由不同的人或小组来构造和连接。由于所有的函数名和标识符都在同一程序里，这就要求所有的开发人员都必须非常小心，不要碰巧使用了相同的函数名和标识符，导致冲突。这种情况很快变得令人烦躁，而且浪费时间，最终导致高昂的代价。

标准的C++有预防这种冲突的机制：**namespace**关键字。库或程序中的每一个C++定义集被封装在一个名字空间中，如果其他的定义中有相同的名字，但它们在不同的名字空间，就不会产生冲突。

名字空间是十分方便和有用的工具，但名字空间的出现意味着在写程序之前，必须知道它们。如果只是简单地包含头文件，使用头文件中的一些函数或对象，编译时，可能会遇到一些奇怪的错误，确切地说，如果仅仅只包含头文件，编译器无法找到任何有关函数和对象的声明。在多次看到编译器的这种提示后，我们会熟悉它所代表的含义（即“虽然包含了头文件，但所有的声明都在一个名字空间中，而没有告诉编译器我们想要用这个名字空间中的声明”）。

可以用一个关键字来声明：“我要使用这个名字空间中的声明和（或）定义”。这个关键字是“**using**”。所有的标准C++库都封装在一个名字空间中，即“**std**”（代表“standard”）。由于本书常使用到这些标准的库文件，几乎在每个程序中都有如下的使用指令（**using**指令）：

```
using namespace std;
```

这意味着打开**std**名字空间，使它的所有名字都可用。有了这条语句，就不用担心特殊的库组件是在一个名字空间中，因为在使用**using**指令地方，它使名字空间在整个文件中都是可用的。

在人们费尽心机把名字空间的名字隐藏起来之后，再暴露名字空间的所有名字，这看起来是矛盾的，而事实上，应该对这样的轻率作法倍加小心（这将在本书后面解释）。但是，**using**指令仅暴露当前文件的名字，所以，它并不像起初听起来那样严重（但是，如果再想一想在头文件中这样做，这就是鲁莽的举动）。

名字空间和包含头文件的方法之间存在着相互关系。现代头文件的包含命令已标准化了（如**<iostream>**，不带扩展名“**.h**”），过去典型包含头文件的方式是带上“**.h**”，如**<iostream.h>**。那时，名字空间不是语言的一部分。所以，对已经存在的代码要提供向后兼

容，如果给出

```
#include <iostream.h>
```

它相当于

```
#include <iostream>
using namespace std;
```

但本书使用标准的包含格式（即不带“.h”），因此就必须显式地使用**using**指令。到此，介绍了对名字空间必须了解的内容，在第10章将更全面地讨论这个问题。

### 2.3.3 程序的基本结构

C/C++程序是变量、函数定义、函数调用的集合。程序开始运行时，它执行初始化代码并调用一个特殊的函数“**main()**”。程序的主要代码放在这里。

正如前面提到的，函数定义包含返回类型（在C++中必须指明）、函数名、括号内的参数列表、大括号内的函数代码。下面是一个简单的定义：

```
int function() {
    // Function code here (this is a comment)
}
```

上面这个函数，参数列表为空，函数体内只含有一条注释。

一个函数定义可有多对花括号，但必须有一对把整个函数体括起来。**main()**也是一个函数，也必须遵守这些规则。在C++语言中，**main()**总是返回**int**类型。

C/C++语言书写格式比较自由。除了个别情况，编译器忽略换行符和空格符，所以，必须有某种方式来判定一条语句的结束。C++语句是以分号结束。

C的注释行以“/\*”开始，以“\*/”结束，其中可以包含换行符。C++可使用C风格的注释，也有自己的注释符：“//”。注释从“//”开始，到换行结束。对于只有一行的注释，比起“/\* \*/”来，“//”要方便得多，本书将较多地使用。

### 2.3.4 “Hello, World!”

现在，终于写出第一个程序：

```
//: C02:Hello.cpp
// Saying Hello with C++
#include <iostream> // Stream declarations
using namespace std;

int main() {
    cout << "Hello, World! I am "
        << 8 << " Today!" << endl;
} ///:~
```

通过“<<”操作符把一系列的参数传递给**cout**对象。然后**cout**对象按从左向右的顺序将参数打印出来。输入输出流函数**endl**表示一行结束并在行末加上一个换行符。使用输入输出流，可将一系列的参数按顺序排起来，使类易于使用。

在C语言中，用双引号括起来的正文称为“字符串”(*string*)。标准的C++类库有一个专门用于正文处理的功能强大的**string**类，所以我们将使用更精确的术语“字符数组”



(*character array*) 来描述双引号之间的正文。

编译器为字符数组分配存储空间, 把每个字符相应的ASCII码存放到这个空间中。编译器在字符数组后自动加上含“0”值的额外存储片, 标志数组结束。

在字符数组内, 通过使用“转义序列”可以插入一些特殊的字符。转义序列是由反斜杠(\) 跟上一个特殊的代码组成。例如, “\n” 意思是换行。编译器手册或是C语言指南给出了一组完整的转义序列, 其他包括“\t”(跳格), “\”(反斜杠), “\b”(空格)。

注意, 一条语句可占多行, 整条语句以分号结束。

字符数组变量和常数混合出现在上述cout语句中。使用cout语句时, 操作符“<<”根据所带的参数以不同的含义重载, 所以当向cout发送不同的参数时, 它能“识别应该对这个参数作何处理”。

本书中, 在每个文件的第一行都有一条注释, 以注释符(一般是“//”)跟一个冒号开始, 而最后一行是一条注释后面跟着一个“/::~”, 表示文件结束。这是一点技巧, 这样标记, 可以很容易地从代码文件中提取信息(本书第2卷中可找到这个提取信息的程序, 该卷在[www.BruceEckel.com](http://www.BruceEckel.com)上)。第一行的注释中有文件名和位置信息, 因此文件能被正文和其他文件引用, 所以很容易从本书的源代码中找到它(源代码可从[www.BruceEckel.com](http://www.BruceEckel.com)下载)。

### 2.3.5 运行编译器

下载并解压缩本书的源代码后, 在子目录CO2下找到这个程序。用Hello.cpp作为参数调用编译器。对于这样一个简单的、单文件程序, 一般的编译器都能很容易地完成它的编译。例如, 用GNU C++ 编译器(它在Internet上可免费获得), 可以输入

```
g++ Hello.cpp
```

其他编译器也有类似的语法, 有关细节可参阅编译器文档。

## 2.4 关于输入输出流

前面所看到的仅仅是输入输出流类最基本的用法。它的输出还有另外的一些格式, 比如, 对于数值的输出格式有十进制、八进制、十六进制。下面是另一个使用输入输出流的例子:

```
//: C02:Stream2.cpp
// More streams features
#include <iostream>
using namespace std;

int main() {
    // Specifying formats with manipulators:
    cout << "a number in decimal: "
         << dec << 15 << endl;
    cout << "in octal: " << oct << 15 << endl;
    cout << "in hex: " << hex << 15 << endl;
    cout << "a floating-point number: "
         << 3.14159 << endl;
    cout << "non-printing char (escape): "
         << char(27) << endl;
} ///::~
```

在这个例子中, 输入输出流利用iostream操作符, 将数字分别以十进制、八进制和十六进

制打印出来（操作符不进行打印操作，但它改变输出流的状态）。浮点数的格式由编译器自动确定。此外，通过（显式）类型转换（*cast*），任何字符都能转换成`char`类型（`char`是保存单字符的数据类型），发送到流对象。显式类型转换看起来像函数调用：`char()`带上字符的ASCII码值。在上述程序中，`char(27)`是把“escape”发送到`cout`。

### 2.4.1 字符数组的拼接

C预处理器的一个重要功能就是可以进行字符数组的拼接（*character array concatenation*）。书中的一些例子要用到这项重要功能。如果两个加引号的字符数组邻接，并且它们之间没有标点，编译器就会把这些字符数组连接成单个字符数组。当代码列表宽度有限制时，字符数组的拼接就特别有用。

```
//: C02:Concat.cpp
// Character array Concatenation
#include <iostream>
using namespace std;

int main() {
    cout << "This is far too long to put on a "
         << "single line but it can be broken up with "
         << "no ill effects\nas long as there is no "
         << "punctuation separating adjacent character "
         << "arrays.\n";
} ///:~
```

初看，上述程序好像是错的，因为在每行结束没有分号。请记住C/C++是自由格式语言，虽然一般情况下看到在每行的末尾带有一个分号，但实际要求是在每个语句结束时才加分号，而一个语句很可能要写好几行。

### 2.4.2 读取输入数据

输入输出流类提供了读取输入的功能。用来完成标准输入功能的对象是`cin`（代表“console input”，控制台输入）。`cin`通常是指从控制台输入，但这种输入可以重定向来自其他输入源。后面将用例子说明。

和`cin`一起使用的输入输出流操作符是“>>”。这个操作符接受与参数类型相同的输入。例如，如果设定了一个整型参数，它将等待从控制台传来的一个整数。下面是一个例子：

```
//: C02:Numconv.cpp
// Converts decimal to octal and hex
#include <iostream>
using namespace std;

int main() {
    int number;
    cout << "Enter a decimal number: ";
    cin >> number;
    cout << "value in octal = 0"
         << oct << number << endl;
    cout << "value in hex = 0x"
         << hex << number << endl;
} ///:~
```

这个程序是将用户输入的数字转换为八进制和十六进制表示。

### 2.4.3 调用其他程序

典型的例子是在Unix shell脚本或DOS批处理文件中，使用从标准输入输出读写的程序。用标准的C语言**system()**函数，C/C++程序可调用任何程序。**system()**函数在头文件**<cstdlib>**中已声明：

```
//: C02:CallHello.cpp
// Call another program
#include <cstdlib> // Declare "system()"
using namespace std;

int main() {
    system("Hello");
} ///:~
```

为了使用**system()**，通常需要在操作系统命令提示下输入字符数组。输入的字符数组可以包含命令行参数，字符数组也可以是运行时产生的（不只是如上面所示的使用静态字符数组）。执行命令字符数组，把控制返回给程序。

从这个程序可以看出，在C++中使用普通的C库函数是很容易的事，只要包含头文件和调用所需的库函数就行了。如果已经学过C语言，那么C与C++向上兼容的特性，会为学习C++带来很大的帮助。

## 2.5 字符串简介

虽然字符数组很有用，但它有一定的限制。简单地说它就是存放在内存中的一组字符，如果要用它做什么事情，必须处理所有细节。例如，引号内字符数组的大小在编译时就确定了。如果想在这样的字符数组中添增字符，需要了解很多有关的知识（包括动态内存管理，字符数组的拷贝、连接等），才能完成添加任务。这正是我们所希望的有一种对象能替我们完成的事。

标准的C++**string**类就是设计用来处理（并隐藏）对字符数组的低级操作，而这些操作早期是由C程序员来完成的。从有C语言以来这些操作就一直是一个编程费时、产生错误的原因。虽然本书第二卷中专门有一章介绍**string**类，但由于**string**能简化编程，对程序编写十分重要，所以，在此对它作一些介绍并加以使用。

为使用**string**类，需要包含C++头文件**<string>**。**string**类在名字空间**std**中，因此要用**using**指令。由于操作符重载，**string**类的使用是很直观的：

```
//: C02:HelloStrings.cpp
// The basics of the Standard C++ string class
#include <string>
#include <iostream>
using namespace std;

int main() {
    string s1, s2; // Empty strings
    string s3 = "Hello, World."; // Initialized
    string s4("I am"); // Also initialized
    s2 = "Today"; // Assigning to a string
```

```

s1 = s3 + " " + s4; // Combining strings
s1 += " 8 "; // Appending to a string
cout << s1 + s2 + "!" << endl;
} ///:~

```

前两个字符串s1和s2开始时是空的。s3和s4的两种不同初始化方法效果是相同的（也可简单地用一个string对象来初始化另一个string对象）。

可以用“=”来给string对象赋值。“=”用其右边的内容代替string对象先前的内容。不必为先前的内容费心，它被自动处理。连接string对象，只需用“+”操作符。“+”也可将string连接到字符数组中。如果想将string加到一个string或字符数组之后，可以用“+=”操作符完成这一操作。最后说明一点，输入输出流知道如何来处理string，所以可直接向cout发送string（或能产生string的表达式，如上面的例子中的s1+s2+“!”）来打印它。

## 2.6 文件的读写

在C语言中，完成打开和处理文件这样复杂的操作，需要对C语言有较深的了解。然而C++语言的iostream库提供了一种简单的方法来处理文件，因此，介绍这个功能可以比在C语言中介绍这一功能更早。

为了打开文件进行读写操作，必须包含<fstream>。虽然<fstream>会自动包含<iostream>，但如果打算使用cin，cout，最好还是显式地包含<iostream>。

为了读而打开文件，要创建一个ifstream对象，它的用法与cin相同，为了写而打开文件，要创建一个ofstream对象，用法与cout相同。一旦打开一个文件，就可以像处理其他iostream对象那样对它进行读写，非常简单。

在iostream库中，一个十分有用的函数是getline()，用它可以把一行读入到string对象中（以换行符结束）<sup>①</sup>。getline()的第一个参数是ifstream对象，从中读取内容，第二个参数是stream对象。函数调用完成之后，string对象就装载了一行内容。

下面是一个简单的例子，将一个文件的内容拷贝到另一个文件：

```

//: C02:Scopy.cpp
// Copy one file to another, a line at a time
#include <string>
#include <fstream>
using namespace std;

int main() {
    ifstream in("Scopy.cpp"); // Open for reading
    ofstream out("Scopy2.cpp"); // Open for writing
    string s;
    while(getline(in, s)) // Discards newline char
        out << s << "\n"; // ... must add it back
} ///:~

```

从上面的程序可以看出，为了打开一个文件，只要将欲建立的文件名交给ifstream和ofstream对象即可。

这里引入了一个新概念——while循环。我们将在下一章对它进行详细的介绍。while循环

<sup>①</sup> getline()实际有很多参数，我们将在第2卷的“iostreams”一章中详尽讨论。

的基本思想是用**while**后面带括号中的表达式来控制下一条句（也可以是用大括号括起来的多条语句）的执行。只要括号中的表达式（在这个例子中是**getline(in,s)**）产生“true”结果，则继续执行由**while**控制的语句。就是说，如果**getline()**成功地读入一行，它就返回“true”值。如果到达输入结束，则返回“false”。上面程序中**while**循环逐行读取输入文件，然后将它们写入到输出文件。

**getline()**逐行读取字符，遇到换行符终止（终止字符是可以改变的，我们在第二卷输入输出流一章再讨论）。**getline()**将丢弃换行符而不把它存入**string**对象。因此，想使拷贝的文件看上去和源文件一样，必须加上换行符，如上所示。

另一个有趣味的例子是把整个文件拷贝成单独的一个**string**对象：

```
//: C02:FillString.cpp
// Read an entire file into a single string
#include <string>
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    ifstream in("FillString.cpp");
    string s, line;
    while(getline(in, line))
        s += line + "\n";
    cout << s;
} ///:~
```

**string**具有动态特性，不必担心**string**的内存分配；只管添加新内容进去就行了，**string**会自动扩展以保存新的输入。

把整个文件都输入到一个字符串中，好处之一就是，**string**类有许多函数可用来对字符串进行查找和操作，使用它们可以把文件当成单个的字符串来处理。但也有一定的局限性。把一个文件作为许多行的集合而不是一大段文本来处理，通常是很方便的。例如，如果想对每一行都加上行号，把每行作为一个单独的**string**对象会非常容易。要完成这项工作，我们需用别的方法。

## 2.7 vector简介

使用**string**，我们可以向**string**对象输入数据而不关心需要多少存储空间。但如果把每一行读入一个**string**对象，我们就不知道需要多少**string**——只有读完整个文件后才知道。为了解决这一问题，我们需要有某种能够自动扩展的存放设施，用以包含所需数量的**string**对象。

实际上，为什么要限制我们自己只存放**string**对象呢？当编写程序时，很多情况下并不知道会用到多少什么东西。如果有某种“容器”对象，它能容纳所有的各种对象，这似乎更有用。幸运的是，标准C++库有一个现成的解决方法：标准容器（container）类。容器类是标准C++非常实用的强大工具之一。

人们经常会把标准C++库的“容器”与“算法”和被称为STL的东西相混淆。STL（标准模板类库，Standard Template Library）是1994年春天Alex Stepanov在加州San Diego的会议上把他的C++库提交给C++标准委员会时使用的名称（Alex Stepanov当时在惠普公司工作）。这个名称一直沿用下来，特别是惠普决定允许这个库公开下载后，使用的人就更多了。同时，

C++标准委员会对STL作了大量的修改，将它整合进标准C++类库。SGI公司(参见<http://www.sgi.com/Technology/STL>)不断对STL进行改进。SGI的STL与标准的C++库在许多细节上是不同的。虽然人们经常产生误解，但实际上C++标准是不“包括”STL的。由于标准C++库的“容器”和“算法”与SGI的STL有相同的来源(通常同名)，因此容易引起误会。所以，本书中，将使用“标准C++库”或“标准库容器”或其他类似的说法，避免使用“STL”这个术语。

虽然标准C++库容器和算法的实现所使用的某些概念较深奥，并且在本书第2卷中专门用了两章来讲解这些概念，但即使对这些概念不太了解，也不妨碍这些库的使用。最基本的标准容器——“**vector**”非常有用，在这里对它作一些介绍，以后会经常用到。我们会发现，使用**vector**后，可以进行大量的工作而不用关心底层的实现(再强调一下，这就是面向对象编程的一个重要目标)。当读完第2卷中有关标准类库的章节后，我们会学到更多的关于**vector**和其他容器的知识。如果本书较早的程序中使用**vector**并不像有经验的C++程序员所做的那样，这是可以理解的。一般说来，这里的多数用法还是适当的。

**vector**类是一个模板(*template*)，也就是说它可有效地用于不同的类型。就是说，我们可以创建**Shape**的**vector**、**Cat**的**vector**和**String**的**vector**等。用模板几乎可以创建“任何事物的类”。把类型名输入到尖括号内，让编译器知道**vector**所用的类(在这种情况下就是**vector**将要保存的类)。所以，**string**的**vector**表示为**vector<string>**。这样，就定制了只装**string**对象的**vector**。如果试图在这个**vector**中加入其他类型，编译器会给出错误提示信息。

既然**vector**表达了“容器”的概念，就应该有一定的方法把东西放进容器中，并且能从容器里把东西取出来。为了在**vector**末尾后追加一个新元素，可以使用成员函数**push\_back()**(注意，对于一个具体的对象要用“.”号来调用它的成员函数)。“**push\_back()**”这个名字看上去似乎有些冗长，不如“**put**”简单，这样命名是因为还有别的容器和成员函数也要向容器添加新元素。例如，**insert()**成员函数，它是在容器中间加入新元素，**vector**支持这个函数，但它的用法更复杂，第2卷再解释它。还有**push\_front()**函数(不属于**vector**)，它是把新元素加到**vector**的开头。在**vector**中，还有很多成员函数，在标准的C++类库中，还有很多容器，但是令人惊奇的是，仅仅知道一些简单的特征就能做许多事情了。

可以用**push\_back()**向**vector**内添加新元素，但怎样从**vector**取回这些元素呢？解决的方法很巧妙——操作符重载，让**vector**像数组那样使用。几乎每一种编程语言都有数组这种数据类型(下一章将对它作更多的讨论)。数组是一个集合体，即它由许多元素构成。数组的一个显著特点是它所有的元素大小相同且逐个邻接。最重要的是元素可由“下标”(indexing)选定，这意味着，只要说“我要第n个元素”，就能找到这个元素，通常很快。除了某些特例，一般的编程语言下标都用方括号表示。比如，对于一个数组**a**，想提出第5个单元，就可以写成**a[4]**(注意下标总是从0开始)。

正如“<<”和“>>”可用于**iostreams**类一样，通过操作符重载也可把简单有效的下标记号用于**vector**类中。不必知道重载是如何实现的——它留到下一章讨论——但是，如果知道为了使用[]与**vector**一起操作而隐藏了的某些技巧，这对于加深理解是有帮助的。

了解了上述内容，现在来看一个使用**vector**的程序。为使用**vector**，必须包含头文件**<vector>**：

```
//: C02:Fillvector.cpp
```



```
// Copy an entire file into a vector of string
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

int main() {
    vector<string> v;
    ifstream in("Fillvector.cpp");
    string line;
    while(getline(in, line))
        v.push_back(line); // Add the line to the end
    // Add line numbers:
    for(int i = 0; i < v.size(); i++)
        cout << i << ": " << v[i] << endl;
} ///:~
```

程序大部分与前一个程序相同，打开文件并每次将一行读进 **string** 对象。不同的是，这些**string**对象被压入**vector v**的尾部。**while**循环完成时，整个文件存在于**v**内，并驻留内存。

**while**语句之后是**for**循环语句。它与**while**语句相似，不过它多了一些控制条件。**for**之后的括号内是控制表达式，这和**while**语句相同。但它有一个在括号内的控制表达式，由三部分组成：第一部分初始化；第二部分检测退出循环的条件；第三部分改变某些内容，通常是为了遍历一个数据项序列。程序中的这种**for**循环方式是非常通行的用法：初始化部分**int i=0**表示用一个整数*i*作循环计数器，并初始化为0；检测部分表明，要使循环继续，*i*的值必须小于**vector**对象**v**中的元素个数（元素个数由成员函数**size()**得出）；最后一部分用到了C/C++中的自增操作符，使*i*加1。确切地说，**i++**表示取*i*的值加上1，并把结果返回给*i*。所以，整个**for**循环就是取控制变量*i*，使它从0逐渐递增至比**vector**对象的个数小1时结束。对于*i*的每一个值，执行一次**cout**语句，建立一行，它由*i*的值（由**cout**转化为字符数组）、分号、空格、文件中的一行句和由**endl**产生的一个换行符组成。编译和运行这个程序，可以看出其结果是给文件加上了行号。

因为在**iostreams**中能使用操作符“>>”，所以可以很容易地修改上面的程序，使之把输入分解成由空格分隔的单词而不是一些行。

```
//: C02:GetWords.cpp
// Break a file into whitespace-separated words
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

int main() {
    vector<string> words;
    ifstream in("GetWords.cpp");
    string word;
    while(in >> word)
        words.push_back(word);
    for(int i = 0; i < words.size(); i++)
        cout << words[i] << endl;
} ///:~
```

表达式:



```
while(in >> word)
```

意思是每次取输入的一个单词，当表达式的值为“false”时，就意味着文件读完了。当然，以空白来分隔单词是比较原始的办法，这里只是举一个简单例子。后面，会看到更复杂的例子，它们可以根据任何方式分割输入。

为了进一步说明使用可带任何类型的**vector**是很容易的事，下面给出一个创建**vector<int>**的例子：

```
//: C02:Intvector.cpp
// Creating a vector that holds integers
#include <iostream>
#include <vector>
using namespace std;

int main() {
    vector<int> v;
    for(int i = 0; i < 10; i++)
        v.push_back(i);
    for(int i = 0; i < v.size(); i++)
        cout << v[i] << ", ";
    cout << endl;
    for(int i = 0; i < v.size(); i++)
        v[i] = v[i] * 10; // Assignment
    for(int i = 0; i < v.size(); i++)
        cout << v[i] << ", ";
    cout << endl;
} ///:~
```

创建可以存放不同类型的**vector**，只需把类型当做模板参数（即在尖括号中的参数）输入即可。提供模板和设计完善的模板库正是为了使这种使用变得容易。

在这个例子中，我们还可以看到**vector**的另外一个重要特征。在表达式

```
v[i] = v[i] * 10;
```

中可以看到，**vector**不仅仅限于输入和取出，还可以通过使用方括号的下标操作符向**vector**的任何一个单元赋值（从而改变单元的值）。这说明**vector**是通用、灵活的“暂存器”，用来处理对象集。在后面几章我们将充分地利用它。

## 2.8 小结

本章主要说明，如果有人已经定义了我们所需要的类，则面向对象编程是很容易的事。这时，只需简单地包含一个头文件，创建对象，并向对象发送消息。如果所用的类功能很强而且设计完善，那么我们不需费很多的力气就能编写出很好的程序。

在显示使用库类使面向对象编程变得简单的过程中，本章也介绍了标准C++库中一些最基本的和十分有用的类型：一系列的输入输出流（特别是从文件和控制台进行读写的输入输出流）、**string**类和**vector**模板。可以看到使用这些库类是多么简单。现在可以想象用它们来编写程序完成许多工作，实际上，它们能做更多的事情<sup>①</sup>。虽然本书的前几章只用了这些工具

<sup>①</sup> 如果读者急于了解这些或者其他标准类库组件的功能，参见[www.BruceEckel.com](http://www.BruceEckel.com)和[www.dinkumware.com](http://www.dinkumware.com)上的本书第2卷。

很少的一部分功能，但对于用C这样的低级语言的编程方式已经是迈出了一大步。学习C语言的低层方面是为了教学目的，同时也很费时。如果用对象来管理低层的事务，最终会更有效。毕竟，面向对象编程就是要隐藏具体的细节，使我们着眼于程序设计更大的方面。

尽管面向对象编程尽可能使编程工作在较高的层次上进行，但C语言的某些基本知识是不能不知道的，这些将在第3章中讨论。

## 2.9 练习

部分练习题的答案可以在本书的电子文档“*Thinking in C++ Annotated Solution Guide*”中找到，只需支付很少的费用就可以从<http://www.BruceEckel.com>获得这个电子文档。

- 2-1 修改**Hello.cpp**，使它能打印你的名字和年龄（或者你的鞋码、爱犬的年龄等，只要你喜欢）。编译并运行修改后的程序。
- 2-2 以**Stream2.cpp**、**Numconv.cpp**为例，编一个程序，让它根据输入的半径值求出圆面积，并打印。可以用运算符“\*”求半径的平方。注意，不要用八进制或十六进制格式打印（它们只适用于整数类型）。
- 2-3 编一个程序用来打开文件并统计文件中以空格隔开的单词数目。
- 2-4 编一个程序统计文件中特定单词的出现次数（要求使用**string**类的运算符“==”来查找单词）。
- 2-5 修改**Fillvector.cpp**使它能从后向前打印各行。
- 2-6 修改**Fillvector.cpp**使它能将**vector**中的所有元素连接成单独的一个字符串，并打印，但不要加上行号。
- 2-7 编一个程序，一次显示文件的一行，然后，等待用户按回车键后显示下一行。
- 2-8 创建一个**vector<float>**，并用一个**for**循环语句向它输入25个浮点数，显示**vector**的结果。
- 2-9 创建三个**vector<float>**对象，与第8题一样填写前两个对象。编一个**for**循环，把前两个**vector**的每一个相应元素相加起来，结果放入第三个**vector**的相应元素中。显示这三个**vector**的结果。
- 2-10 编一个程序，创建一个**vector<float>**，像前面的练习那样输入25个数。求每个数的平方，并把它们放入**vector**的同样位置。显示运算前后的**vector**。



## C++中的C

因为C++是以C为基础的，所以要用C++编程就必须熟悉C的语法，就像要解决微分问题必须要对代数十分了解一样。

如果读者以前从没有接触过C，本章将会提供在C++中使用C风格的一个很好的背景知识。如果读者对Kernighan & Ritchie所著的C语言书（经常称之为K&R C）第1版中描述的C风格比较熟悉的话，就会发现在C++以及在标准C中有一些新的、不一样的特征。如果读者对标准C熟悉的话，则应当通览本章找出C++中与众不同的特点。注意这里介绍的是C++的一些基本特征，这些特征和C的特征很相似或者是对C进行的一些修改。C++的更为复杂的特征将会在后面各章中介绍。

本章结合读者对其他语言编程的经验，对C的构造和C++的一些基本结构作了简要介绍。更为详细的介绍参见本书的附带光盘，“*Thinking in C: Foundations for Java & C++*”（Chuck Allison著，MindView公司出版，也可以在[www.MindView.net](http://www.MindView.net)上得到）。这是一个在光盘上的讲座，其目的是让读者仔细地浏览C语言的基础知识。它着重于从C转向使用C++或Java语言所必需的知识，而不是试图让读者成为懂得C的所有细节的专家（使用C++或Java这样的高级语言的一个原因正是由于它们可以避免涉及许多这样的细节）。它也包括练习和解答指南。记住，光盘的内容不能代替本章，而只能作为本章和本书的准备知识，因为本章超出了这张光盘所包含的内容。

### 3.1 创建函数

在旧版本（标准化之前）的C中，我们可以用带任意个数和类型的参数调用函数，编译器都不会报告出错。运行程序之前每一件事情似乎都很好，但当运行时，我们却会得到一些奇怪的结果（更糟的是程序崩溃），并且没有说明为什么会这样的任何提示。缺乏对参数传递的协助以及会导致高深莫测的故障，可能是C被称为“高级汇编语言”的一个原因。在C标准化之前，程序员只能去适应这种情况。

标准C和C++有一个特征叫做函数原型（function prototyping）。用函数原型，在声明和定义一个函数时，必须使用参数类型描述。这种描述就是“原型”。调用函数时，编译器使用原型确保正确传递参数并且正确地处理返回值。如果调用函数时程序员出错了，编译器就会捕获这个错误。

实际上，在前一章中已经学习了函数原型（但并没有这样命名），因为在C++中函数声明的形式需要正确的原型。在函数原型中，参数表包含了应当传递给函数的参数类型和参数的标识符（对声明而言可以是任选的）。参数的顺序和类型必须在声明、定义和函数调用中相匹配。下面是一个声明函数原型的例子：

```
int translate(float x, float y, float z);
```

在函数原型中声明变量时，不能使用和定义一般变量同样的形式。就是说不能用**float x, y, z**。必须指明每一个参数的类型。在函数声明中，下面的形式是可以接受的：

```
int translate(float, float, float);
```

因为在调用函数时，编译器只是检查类型，所以使用标识符只是为了使别人阅读代码时更加清晰。

在函数定义中，因为参数是在函数内部引用的，所以需要命名。

```
int translate(float x, float y, float z) {
    x = y = z;
    // ...
}
```

这条规则只应用于C。在C++中，函数定义的参数表中可以使用未命名的参数。当然，因为它没有被命名，所以不能在函数体中使用它。允许不命名参数是为了给程序员提供在“参数列表中保留位置”的一种方式。不管谁调用函数都必须使用正确的参数。但是，创建函数的人将来可以使用这个参数，而不需要强制修改调用这个函数的代码。即使给出命名，在参数表中忽略这个参数也是可能的，但每次编译函数时，会得到这个值没有被使用这样一条令人讨厌的警告消息。如果删除这个名字，这个警告也会消除。

C和C++有两种其他声明参数列表的方式。如果有一个空的参数列表，可以在C++中声明这个函数为`func()`，它告诉编译器，这里有0个参数。应该意识到这并不意味着在C++中是空参数列表。在C中，它意味着不确定的参数数目（这是C中的漏洞，因为在这种情况下不能进行类型检查）。在C和C++中，声明`func(void)`都意味着空的参数列表。在这种情况下`void`这个关键词意味着“空”（在本章的后面将会看到，就指针而言它也可以表示“没有类型”）。

在不知道会有多少个参数或什么样类型的参数时，参数表的另一种选择是可变的参数列表。这个“不确定参数列表”用省略号（`...`）表示。定义一个带可变参数列表的函数比定义一个带固定参数列表的函数要复杂得多。如果（因为某种原因）不想使用函数原型的错误检查功能，可以对有固定参数表的函数使用可变参数列表。正因为如此，应该限制对C使用可变参数列表并且在C++中避免使用（正如我们将会看到的，在C++中有更好的选择）。在你的C指南的库部分对使用可变参数列表做了描述。

### 3.1.1 函数的返回值

C++函数原型必须指明函数的返回值类型（在C中，如果省略返回值，表示默认为整型）。返回值的类型放在函数名的前面。为了表明没有返回值可以使用`void`关键字。如果这时试图从函数返回一个值会产生错误。下面有一些完整的函数原型：

```
int f1(void); // Returns an int, takes no arguments
int f2(); // Like f1() in C++ but not in Standard C!
float f3(float, int, char, double); // Returns a float
void f4(void); // Takes no arguments, returns nothing
```

要从一个函数返回值，我们必须使用`return`语句。`return`语句退出函数返回到函数调用后的那一点。如果`return`有参数，那个参数就是函数的返回值。如果函数规定返回一个特定类型的值，那么每一个`return`语句都必须返回这个类型。在一个函数定义中可以有多个`return`语句。

```
//: C03:Return.cpp
// Use of "return"
#include <iostream>
using namespace std;
```

```

char cfunc(int i) {
    if(i == 0)
        return 'a';
    if(i == 1)
        return 'g';
    if(i == 5)
        return 'z';
    return 'c';
}

int main() {
    cout << "type an integer: ";
    int val;
    cin >> val;
    cout << cfunc(val) << endl;
} ///:~

```

在函数`cfunc()`中，第一个值为真的`if`语句，通过`return`语句退出函数。注意函数声明不是必须的，因为函数在`main()`使用它之前定义，所以编译器从函数定义中知道它。

### 3.1.2 使用C的函数库

用C++编程时，当前C函数库中的所有函数都可以使用。在定义自己的函数之前，应该仔细地看一下函数库，可能有人已经解决了我们的问题，而且进行了更多的思考和调试。

注意，尽管很多编译器包含大量的额外函数可以使编程更加容易、吸引大家去使用，但是这并不是标准C库的一部分。如果我们肯定不想移植该应用程序到别的平台上（谁又能肯定呢？），那么就使用那些函数，让编程更加容易。如果希望该应用程序具有可移植性，就应该限制使用标准库函数。如果必须执行特定平台的活动，应当尽力把代码隔离在某一场所，以便移植到另一平台时容易进行修改。C++中，经常把特定平台的活动封装在一个类中，这是一个理想的解决办法。

使用库函数的方法如下：首先，在编程参考资料中查找函数（很多编程参考资料按字母顺序排序函数）。函数的描述应该包括说明代码语法的部分。这部分的头部通常至少有一行`#include`，表示包含函数原型的头文件。在程序文件中复制这个`#include`行，所以能正确声明函数。现在可以按照函数出现在语法部分的同样方式来调用它。如果出错了，编译器通过把函数调用和头文件中的函数原型相比较来报告错误。连接器通过默认路径查找标准库，所以在编程时需要做的就是包含这个头文件和调用这个函数。

### 3.1.3 通过库管理器创建自己的库

我们可以将自己的函数收集到一个库中。大多数编程包带有一个库管理器来管理对象模块组。每一个库管理器有它自己的命令，但有这样一个共同的想法：如果想创建一个库，那么就建立一个头文件，它包含库中的所有函数原型。把这个头文件放置在预处理器搜索路径中的某处，或者在当前目录中（以便能被`#include`“头文件”发现），或者在包含路径中（以便能被`#include<头文件>`发现）。现在把所有的对象模块连同建成后的库名传递给库管理器（大多数库管理器要求有一个共同的扩展名，例如`.lib`或`.a`）。把建成的库和其他库放在同一个位置以便连接器能发现它。当使用自己的库时，必须向命令行添加一些东西，让连接器知



道为你调用的函数查找库。因为函数库随着系统而异，所以必须在你的系统手册中查找所有的细节。

## 3.2 执行控制语句

本节涵盖了C++中的执行控制语句。在读写C或C++代码之前，必须熟悉这些语句。

C++使用C的所有执行控制语句。这些语句包括**if-else**、**while**、**do-while**、**for**和**switch**选择语句。C++也允许使用声名狼藉的**goto**语句，在本书中会避免使用它。

### 3.2.1 真和假

所有的条件语句都使用条件表达式的真或假来判定执行路径。**A == B**是一个条件表达式的例子。这里使用条件运算符“**==**”确定变量**A**是否等于变量**B**。表达式产生布尔值**true**（真）或**false**（假）（这只是C++中的关键字，在C中如果一个表达式等于非零值则为“真”）。其他的条件运算符有：**>**、**<**、**>=**等。条件语句在本章的后面会有更详细的介绍。

### 3.2.2 if-else语句

**if-else**语句有两种形式：用**else**或不用**else**。这两种形式是：

```
if (表达式)
    语句
```

或

```
if (表达式)
    语句
else
    语句
```

“表达式”的值为真或假。“语句”是以一个分号结束的简单语句，或一组包含在大括号里的简单语句构成的一个复合语句。不管什么时候使用“语句”，都意味着是简单语句或复合语句。注意这个语句也可能是另一个**if**语句，所以它们可连成一串。

```
//: C03:Ifthen.cpp
// Demonstration of if and if-else conditionals
#include <iostream>
using namespace std;

int main() {
    int i;
    cout << "type a number and 'Enter'" << endl;
    cin >> i;
    if(i > 5)
        cout << "It's greater than 5" << endl;
    else
        if(i < 5)
            cout << "It's less than 5 " << endl;
        else
            cout << "It's equal to 5 " << endl;

    cout << "type a number and 'Enter'" << endl;
    cin >> i;
```



```

if(i < 10)
    if(i > 5) // "if" is just another statement
        cout << "5 < i < 10" << endl;
    else
        cout << "i <= 5" << endl;
    else // Matches "if(i < 10)"
        cout << "i >= 10" << endl;
} ///:~

```

缩进控制流语句体是一种习惯用法，以便读者可以很方便地知道它的起点和终点<sup>①</sup>。

### 3.2.3 while语句

**while**、**do-while**和**for**语句是循环控制语句。一个语句重复执行直到控制表达式的计值为假。**while**循环的形式是：

```

while (表达式)
    语句

```

循环一开始就对表达式进行计算。并在每次重复执行语句之前再次计算。

下面的例子一直在**while**循环体内执行，直到输入密码或按control-C键。

```

//: C03:Guess.cpp
// Guess a number (demonstrates "while")
#include <iostream>
using namespace std;

int main() {
    int secret = 15;
    int guess = 0;
    // "!=" is the "not-equal" conditional:
    while(guess != secret) { // Compound statement
        cout << "guess the number: ";
        cin >> guess;
    }
    cout << "You guessed it!" << endl;
} ///:~

```

**while**语句的条件表达式并不仅限于像上面的例子那样只进行一个简单的测试；它也可以像我们希望的那样复杂，只要能产生一个真或假的结果。我们甚至会看到没有循环体而只有一个分号代码：

```

while(/* Do a lot here */)
    ;

```

在这样的情况下，程序员写的条件表达式既进行循环条件测试，又实现了具体任务。

### 3.2.4 do-while语句

**do-while**的形式是：

```

do
    语句
while (表达式)

```

① 注意，在出现某种缩排约定后，所有的习惯用法都将终结。代码格式风格之间的争执是无休止的。对本书编码风格的描述请看附录A。

**do-while**语句与**while**语句的区别在于，即使表达式第一次计值就等于假，前面的语句也会至少执行一次。在一般的**while**语句中，如果条件第一次为假，语句一次也不会执行。

如果在程序**Guess.cpp**中使用**do-while**，变量**guess**不需要初始为0值，因为它被检测之前就被**cin**语句初始化了：

```
//: C03:Guess2.cpp
// The guess program using do-while
#include <iostream>
using namespace std;

int main() {
    int secret = 15;
    int guess; // No initialization needed here
    do {
        cout << "guess the number: ";
        cin >> guess; // Initialization happens
    } while(guess != secret);
    cout << "You got it!" << endl;
} ///:~
```

因为某种原因，大多数程序员更喜欢只使用**while**语句而避免使用**do-while**语句。

### 3.2.5 for语句

在第一次循环前，**for**循环执行初始化。然后它执行条件测试，并在每一次循环结束时执行某种形式的“步进”。**for**循环的形式是：

```
for(initialization; conditional; step)
    语句
```

表达式中的*initialization*、*conditional*或*step*都可能为空。一旦进入**for**循环，*initialization*代码就执行。在每一次循环之前，*conditional*被测试（如果它的计值一开始就为假，语句就不会执行）。每一次循环结束时，执行*step*。

**for**循环通常用于“计数”任务：

```
//: C03:Charlist.cpp
// Display all the ASCII characters
// Demonstrates "for"
#include <iostream>
using namespace std;

int main() {
    for(int i = 0; i < 128; i = i + 1)
        if (i != 26) // ANSI Terminal Clear screen
            cout << " value: " << i
                << " character: "
                << char(i) // Type conversion
                << endl;
} ///:~
```

读者也许会注意到，变量*i*是在使用它的地方定义，而不是在‘{’所标注的程序块起始处定义。这和传统的过程语言（包括C）形成了对照，过程语言要求在程序块的起始处定义所有的变量。这将在本章的后面讨论。

### 3.2.6 关键字break和continue

在任何一个while、do-while或for循环的结构体中，都能够使用break和continue控制循环的流程。break语句退出循环，不再执行循环中的剩余语句。continue语句停止执行当前的循环，返回到循环的起始处开始新一轮循环。

作为break和continue语句的一个例子，下面程序是一个非常简单的菜单系统：

```
//: C03:Menu.cpp
// Simple menu program demonstrating
// the use of "break" and "continue"
#include <iostream>
using namespace std;

int main() {
    char c; // To hold response
    while(true) {
        cout << "MAIN MENU:" << endl;
        cout << "l: left, r: right, q: quit -> ";
        cin >> c;
        if(c == 'q')
            break; // Out of "while(1)"
        if(c == 'l') {
            cout << "LEFT MENU:" << endl;
            cout << "select a or b: ";
            cin >> c;
            if(c == 'a') {
                cout << "you chose 'a'" << endl;
                continue; // Back to main menu
            }
            if(c == 'b') {
                cout << "you chose 'b'" << endl;
                continue; // Back to main menu
            }
            else {
                cout << "you didn't choose a or b!"
                    << endl;
                continue; // Back to main menu
            }
        }
        if(c == 'r') {
            cout << "RIGHT MENU:" << endl;
            cout << "select c or d: ";
            cin >> c;
            if(c == 'c') {
                cout << "you chose 'c'" << endl;
                continue; // Back to main menu
            }
            if(c == 'd') {
                cout << "you chose 'd'" << endl;
                continue; // Back to main menu
            }
            else {
                cout << "you didn't choose c or d!"
                    << endl;
                continue; // Back to main menu
            }
        }
    }
}
```



```

    }
}
cout << "you must type l or r or q!" << endl;
}
cout << "quitting menu..." << endl;
} ///:~

```

如果用户在主菜单中选择 ‘q’，则用关键字**break**退出，选择其他，程序则继续执行。在每一个子菜单选择后，关键字**continue**用于跳转到while循环的起始处。

**while(true)**语句等价于“永远执行这个循环”。当用户按 ‘q’ 时，**break**语句使程序跳出这个无限循环。

### 3.2.7 switch语句

**switch**语句根据一个整型表达式的值从几段代码中选择执行。它的形式是：

```

switch(selector) {
    case integral-value1 : statement; break;
    case integral-value2 : statement; break;
    case integral-value3 : statement; break;
    case integral-value4 : statement; break;
    case integral-value5 : statement; break;
    (...)
    default: statement;
}

```

选择器 (*selector*) 是一个产生整数值的表达式。**switch**语句把选择器 (*selector*) 的结果和每一个整数值 (*integral-value*) 比较。如果发现匹配，就执行对应的语句 (简单语句或复合语句)。如果都不匹配，则执行**default**语句。

读者也许会注意到上面定义中的每一个**case**后面都以一个**break**语句作为结束，这个**break**语句使得执行跳转到**switch**语句体的结束处 (完成**switch**的闭括号处)。这是建立**switch**语句的一种常用方式，但是**break**是可选的。如果省略它，**case**语句会顺序执行它后面的语句。也就是说，执行后面的各**case**语句代码，直到遇到一个**break**语句。尽管一般不需要这种举动，但是对于一个有经验的程序员来说这可能是有用的。

**switch**语句是一种清晰的实现多路选择的方式 (即对不同的执行路径进行选择)，但它需要一个能在编译时求得整数值的选择器。例如，如果想使用一个字符串类型的对象作为一个选择器，在**switch**语句中它是不能用的。对于字符串类型的选择器，必须使用一系列**if**语句并比较在条件中的字符串。

上面的菜单程序提供了一个特别好的**switch**语句例子：

```

//: C03:Menu2.cpp
// A menu using a switch statement
#include <iostream>
using namespace std;

int main() {
    bool quit = false; // Flag for quitting
    while(quit == false) {
        cout << "Select a, b, c or q to quit: ";
        char response;
        cin >> response;
    }
}

```

```

switch(response) {
    case 'a' : cout << "you chose 'a'" << endl;
               break;
    case 'b' : cout << "you chose 'b'" << endl;
               break;
    case 'c' : cout << "you chose 'c'" << endl;
               break;
    case 'q' : cout << "quitting menu" << endl;
               quit = true;
               break;
    default  : cout << "Please use a,b,c or q!"
               << endl;
}
}
} ///:~

```

**quit** (退出) 标志是**bool** (boolean的简写) 型的, 这种类型只有在C++中才会看到。它只能有**true**或**false**值。选择 'q' 即设置**quit**标志为**true**。下一次计算选择器的值, **quit == false** 返回**false**, 所以不执行**while**循环体。

### 3.2.8 使用和滥用goto

因为关键字**goto**存在于C中, 所以C++中也支持它。使用**goto**经常被贬斥为一种糟糕的编程方式, 大多数时候确实如此。想使用**goto**语句时, 查一下程序代码, 看是否有其他的解决方法。在少数情况下, 可能会发现**goto**语句能够解决用别的方法不能解决的问题, 但是尽管如此, 还应仔细考虑一下。下面是一个例子, 可能会作出似乎有理性的选择:

```

//: C03:gotoKeyword.cpp
// The infamous goto is supported in C++
#include <iostream>
using namespace std;

int main() {
    long val = 0;
    for(int i = 1; i < 1000; i++) {
        for(int j = 1; j < 100; j += 10) {
            val = i * j;
            if(val > 47000)
                goto bottom;
            // Break would only go to the outer 'for'
        }
    }
    bottom: // A label
    cout << val << endl;
} ///:~

```

一个可供选择的方法是设置一个布尔值, 在外层**for**循环对它进行测试, 然后利用**break**从内层**for**循环跳出。然而, 如果我们有几层**for**语句或**while**语句, 可能会出现困难。

### 3.2.9 递归

递归是十分有趣的, 有时也是非常有用的编程技巧, 凭借递归我们可以在一个函数内部调用该函数。当然, 如果这是所做的全部, 那么会一直调用下去, 直到内存用完, 所以一定



要有一种确定“达到底点”递归调用的方法。在下面的例子中，只要递归到`cat`的值超过‘Z’，递归就“达到底点”：<sup>⊖</sup>

```
//: C03:CatsInHats.cpp
// Simple demonstration of recursion
#include <iostream>
using namespace std;

void removeHat(char cat) {
    for(char c = 'A'; c < cat; c++)
        cout << " ";
    if(cat <= 'Z') {
        cout << "cat " << cat << endl;
        removeHat(cat + 1); // Recursive call
    } else
        cout << "VOOM!!!" << endl;
}

int main() {
    removeHat('A');
} ///:~
```

在`removeHat()`中，只要`cat`的值小于‘Z’，就会在`removeHat()`中调用`removeHat()`，从而实现递归。每次调用`removeHat()`，它的参数比当前的`cat`值增加1，所以参数不断增加。

求解某些具有随意性的复杂问题经常使用递归，因为这时解的具体“大小”不受限制，函数可以一直递归调用，直到问题解决。

### 3.3 运算符简介

我们可以把运算符看做是一种特殊的函数（C++的运算符重载正是以这种方式对待运算符）。一个运算符带一个或更多的参数并产生一个新值。运算符参数和普通的函数调用参数相比在形式上不同，但是作用是一样的。

根据读者以前的编程经验，应该习惯于迄今使用的运算符。任何一种编程语言的加（+）、减和单目减（-）、乘（\*）、除（/）和赋值（=）的概念都有同样的意义。本章后面列举出全部运算符集。

#### 3.3.1 优先级

运算符优先级规定表达式中出现多个不同运算符时计值的运算顺序。C和C++中有具体的规则决定计值顺序。最容易记住的是先乘、除，后加、减。如果一个表达式的运算顺序对我们来说是不清晰的，那么对于任何一个读代码的人来说它都可能是不清晰的，所以应该使用括号使计值次序更加清晰。例如：

```
A = X + Y - 2/2 + Z;
```

与带有一组特定的圆括号的同一语句：

```
A = X + (Y - 2)/(2 + Z);
```

---

⊖ 感谢Kris C. Matson建议这个练习题。

具有完全不同的含义。(令 $X = 1, Y = 2, Z = 3$ , 试算一下结果。)

### 3.3.2 自增和自减

C有不少捷径, 因此C++也有很多捷径。这些捷径使得更易于输入代码, 但有时却不易于阅读。可能C语言的设计者认为如果程序员的眼睛不必浏览大范围的印刷区域, 那么理解一段巧妙的代码可能是比较容易的。

其中一个较好的捷径是自增和自减运算符。经常使用它们去改变循环变量以控制循环执行的次数。

自减运算符是‘--’, 意思是“减小一个单位”。自增运算符是‘++’, 意思是“增加一个单位”。例如, 如果A是一个整数, 则++A等于( $A = A + 1$ )。自增和自减产生一个变量的值作为结果。如果运算符在变量之前出现(即++A), 则先执行运算, 再产生结果值。如果运算符在变量之后出现(即A++), 则产生当前值, 再执行运算。例如:

```
//: C03:AutoIncrement.cpp
// Shows use of auto-increment
// and auto-decrement operators.
#include <iostream>
using namespace std;

int main() {
    int i = 0;
    int j = 0;
    cout << ++i << endl; // Pre-increment
    cout << j++ << endl; // Post-increment
    cout << --i << endl; // Pre-decrement
    cout << j-- << endl; // Post decrement
} ///:~
```

如果我们曾经对“C++”这个名字感到奇怪, 那么现在应该明白了。C++隐含的意思就是“在C上更进一步”。

## 3.4 数据类型简介

在编写程序中, 数据类型 (*data type*) 定义使用存储空间 (内存) 的方式。通过定义数据类型, 告诉编译器怎样创建一片特定的存储空间, 以及怎样操纵这片存储空间。

数据类型可以是内部的或抽象的。内建数据类型是编译器本来能理解的数据类型, 直接与编译器关联。C和C++中的内建数据类型几乎是一样的。相反, 用户定义的数据类型是我们和别的程序员创建的类型, 作为一个类。它们一般被称为抽象数据类型。编译器启动时, 知道怎样处理内建数据类型; 编译器再通过读包含类声明的头文件 (在后面几章我们会了解到这一点) 认识怎样处理抽象数据类型。

### 3.4.1 基本内建类型

标准C的内建类型 (由C++继承) 规范不说明每一个内建类型必须有多少位。规范只规定内建类型必须能存储的最大值和最小值。如果机器基于二进制, 则最大值可以直接转换成容纳这个值所需的最少位数。然而, 例如, 如果一个机器使用二进制编码的十进制 (BCD) 来

表示数字，在机器中容纳每一种数据类型的最大数值的空间是不同的。系统头文件**limits.h**和**float.h**中定义了不同的数据类型可能存储的最大值和最小值（在C++中，一般用**#include <climits>**和**<cfloat>**代替）。

C和C++中有4个基本的内建数据类型，这里的描述是基于二进制的机器。**char**是用于存储字符的，使用最小的8位（一个字节）的存储，尽管它可能占用更大的空间。**Int**存储整数值，使用最小两个字节的存储空间。**float**和**double**类型存储浮点数，一般使用IEEE的浮点格式。**float**用于单精度浮点数，**double**用于双精度浮点数。

如前所述，我们可以在某一作用域的任何地方定义变量，可以同时定义和初始化它们。下面是怎样用这四种基本数据类型定义变量的例子：

```
//: C03:Basic.cpp
// Defining the four basic data
// types in C and C++

int main() {
    // Definition without initialization:
    char protein;
    int carbohydrates;
    float fiber;
    double fat;
    // Simultaneous definition & initialization:
    char pizza = 'A', pop = 'Z';
    int dongdings = 100, twinkles = 150,
        heehos = 200;
    float chocolate = 3.14159;
    // Exponential notation:
    double fudge_ripple = 6e-4;
} ///:~
```

程序的第一部分定义了4种基本数据类型的变量，没有对变量初始化。如果不初始化一个变量，标准会认为没有定义它的内容（通常，这意味着它们的内容是垃圾）。程序的第二部分同时定义和初始化变量（如果可能，最好在定义时提供初始值）。注意常量6e-4中指数符号的使用，意思是“6乘以10的负4次幂”。

### 3.4.2 bool类型与true和false

在**bool**类型成为标准C++的一部分之前，每个人都想使用不同的方法产生类似**bool**类型的行为。这产生了可移植性问题，可能会引入微妙的错误。

标准C++的**bool**类型有两种由内建的常量**true**（转换为整数1）和**false**（转换为整数0）表示的状态。这3个名字都是关键字。此外，一些语言元素也已经被采纳：

元 素	布尔类型的用法
<b>&amp;&amp;    !</b>	带布尔参数并产生 <b>bool</b> 结果
<b>&lt; &gt; &lt;= &gt;= == !=</b>	产生 <b>bool</b> 结果
<b>if, for, while, do</b>	条件表达式转换为 <b>bool</b> 值
<b>?:</b>	第一个操作数转换为 <b>bool</b> 值

因为有很多现存的代码使用整型**int**表示一个标志，所以编译器隐式转换**int**为**bool**（非零值为**true**而零值为**false**）。理想的情况下，编译器会给我们一个警告，建议纠正这种情况。

用++把一个标志设置为真是一种“糟糕的编程风格”。这样做依然是允许的，但受到抵制，意味着在将来的某个时候它可能是不合法的。问题在于从**bool**到**int**做了隐式类型转换，增加了值（可能超过了0和1的正常布尔值的范围），然后再做相反的隐式转换。

指针（本章的后面将会引入）在必要的时候也自动转换成**bool**值。

### 3.4.3 说明符

说明符（specifier）用于改变基本内建类型的含义并把它们扩展成一个更大的集合。有4个说明符：**long**、**short**、**signed**和**unsigned**。

**long**和**short**修改数据类型具有的最大值和最小值。一般的**int**必须至少有**short int**型的大小。整数类型的大小等级是：**short int**、**int**、**long int**。只要满足最小/最大值的要求，所有的大小可以看成是一样的。例如，在64位字的机器上，所有的数据类型都可能是64位的。

浮点数的大小等级是：**float**、**double**和**long double**。“long float”是不合法的类型，也没有**short**浮点数。

**signed**和**unsigned**修饰符告诉编译器怎样使用整数类型和字符的符号位（浮点数总含有一个符号）。**unsigned**数不保存符号，因此有一个多余的位可用，所以它能存储比**signed**数大一倍的正数。**signed**是默认的，只有**char**才一定要使用**signed**，**char**可以默认为**signed**，也可以不默认为**signed**。通过规定**signed char**，可以强制使用符号位。

下面的例子使用**sizeof**运算符显示用字节表示的数据类型的大小，该运算符在本章的后面介绍：

```
//: C03:Specify.cpp
// Demonstrates the use of specifiers
#include <iostream>
using namespace std;

int main() {
    char c;
    unsigned char cu;
    int i;
    unsigned int iu;
    short int is;
    short iis; // Same as short int
    unsigned short int isu;
    unsigned short iisu;
    long int il;
    long iil; // Same as long int
    unsigned long int ilu;
    unsigned long iilu;
    float f;
    double d;
    long double ld;
    cout
        << "\n char= " << sizeof(c)
        << "\n unsigned char = " << sizeof(cu)
        << "\n int = " << sizeof(i)
        << "\n unsigned int = " << sizeof(iu)
        << "\n short = " << sizeof(is)
        << "\n unsigned short = " << sizeof(isu)
```



```

    << "\n long = " << sizeof(il)
    << "\n unsigned long = " << sizeof(ilu)
    << "\n float = " << sizeof(f)
    << "\n double = " << sizeof(d)
    << "\n long double = " << sizeof(ld)
    << endl;
} ///:~

```

要注意，在不同的机器/操作系统/编译器上运行这个程序得到的结果可能是不同的。因为（如前所述）唯一一致的事情是每个不同类型都具有标准中规定的最小值和最大值。

如上所示，当用**short**或**long**改变**int**时，关键字**int**是可选的。

### 3.4.4 指针简介

不管什么时候运行一个程序，都是首先把它装入（一般从磁盘装入）计算机内存。因此，程序中的所有元素都驻留在内存的某处。内存一般被布置成一系列连续的内存位置；我们通常把这些位置看做是8位字节，但实际上每一个空间的大小取决于具体机器的结构，一般称为机器的字长（*word size*）。每一个空间可按它的地址与其他空间区分。为了便于讨论，我们认为所有机器都使用有连续地址的字节从零开始，一直到该计算机的内存的上限。

因为程序运行时驻留内存中，所以程序中的每一个元素都有地址。假设我们从一个简单的程序开始：

```

//: C03:YourPets1.cpp
#include <iostream>
using namespace std;

int dog, cat, bird, fish;

void f(int pet) {
    cout << "pet id number: " << pet << endl;
}

int main() {
    int i, j, k;
} ///:~

```

程序运行的时候，程序中的每一个元素在内存中都占有一个位置。甚至函数也占用内存。我们将会看到，定义什么样的元素和定义元素的方式通常决定元素在内存中放置的地方。

C和C++中有一个运算符会告诉我们元素的地址。这就是‘&’运算符。只要在标识符前加上‘&’，就会得出标识符的地址。可以修改程序**YourPets1.cpp**，用以打印所有元素的地址。修改如下：

```

//: C03:YourPets2.cpp
#include <iostream>
using namespace std;

int dog, cat, bird, fish;

void f(int pet) {
    cout << "pet id number: " << pet << endl;
}

int main() {

```

```

int i, j, k;
cout << "f(): " << (long)&f << endl;
cout << "dog: " << (long)&dog << endl;
cout << "cat: " << (long)&cat << endl;
cout << "bird: " << (long)&bird << endl;
cout << "fish: " << (long)&fish << endl;
cout << "i: " << (long)&i << endl;
cout << "j: " << (long)&j << endl;
cout << "k: " << (long)&k << endl;
} ///:~

```

(**long**)是一种类型转换 (*cast*)。意思是“不要把它看做是原来的类型，而是看做是**long**类型”。这个类型转换不是必须的，但是如果没说的话，地址是以十六进制的形式打印，所以转换为**long**类型会增加一些可读性。

这个程序的结果会随计算机、操作系统和各种其他的因素的不同而变化，但我们总会看到一些有趣的现象。在我的计算机上运行一次的结果如下：

```

f(): 4198736
dog: 4323632
cat: 4323636
bird: 4323640
fish: 4323644
i: 6684160
j: 6684156
k: 6684152

```

现在可以看到在函数**main()**的内部和外部定义的变量存放在不同的区域；当对语言有更多的了解时，就会明白为什么如此。同样，**f()**出现在它自己的区域，在内存中代码和数据一般是分开存放的。

另一个值得注意的有趣的事情是，相继定义的变量在内存中是连续存放的。它们根据各自的数据类型所要求的字节数分隔开。这个例子中只使用了整型数据类型，变量**cat**距离变量**dog** 4个字节，变量**bird**距离变量**cat** 4个字节，等等。所以在这台机器上，一个**int**占4个字节。

这个有趣的实验显示了怎样分配内存，那么利用地址能干什么呢？能做的最重要的事就是，把地址存放在别的变量中以便以后使用。C和C++有一个专门的存放地址的变量类型。这个变量叫做指针 (*pointer*)。

定义指针的运算符和用于乘法的运算符‘\*’是一样的。正如我们将看到的那样，编译器会根据它所在的上下文知道它表示的不是乘法。

定义一个指针时，必须规定它指向的变量类型。可以先给出一个类型名，然后不是立即给出变量的标识符，而是在类型和标识符之间插入一个星号，这就是说“等一等，它是一个指针”。一个指向**int**的指针如下所示：

```
int* ip; // ip points to an int variable
```

把‘\*’和类型联系起来似乎是很明白且易读的，但是事实上可能容易产生错觉。有人可能更倾向于说“整型指针”好像它是一个单独的类型。可是，对于**int**或其他的基本数据类型，可以写成

```
int a, b, c;
```

而对于指针，可能想写成



```
int* ipa, ipb, ipc;
```

C的语法（并由C++语法继承）不允许像这样合乎情理的表达。在上面的定义中，只有**ipa**是一个指针，而**ipb**和**ipc**是一般的**int**（可以认为“\*和标识符结合得更紧密”）。因此，最好是每一行定义一个指针；这样就能得到一个清晰的语法而不会混淆：

```
int* ipa;
int* ipb;
int* ipc;
```

C++编程的一般原则是在定义时进行初始化，事实上这种形式工作得很好。例如，上面的变量并没有初始化为任何一个特定的值，它们所具有的是一些无意义的值。如果写成下面的形式会更好：

```
int a = 47;
int* ipa = &a;
```

现在已经初始化了**a**和**ipa**，**ipa**存放**a**的地址。

一旦有一个初始化了的指针，我们能做的最基本的事就是利用指针来修改它指向的值。要通过指针访问变量，可以使用以前定义指针使用的同样的运算符来间接引用这个指针，如像：

```
*ipa = 100;
```

现在**a**的值是100而不是47。

这些是指针基础：可以保存地址，可以使用地址去修改原先的变量。但还是留下问题：为什么要通过另一个变量作为代理来修改一个变量？

通过对指针的介绍，我们可以把答案分为两大类：

- 1) 为了能在函数内改变“外部对象”。这可能是指针最基本的用途，并且在下一小节对它进行验证。
- 2) 为了获得许多灵活的编程技巧，而这些将在本书的其余部分见到。

### 3.4.5 修改外部对象

通常，向函数传递参数时，在函数内部生成该参数的一个拷贝。这称为按值传递（*pass-by-value*）。在下面的程序中能看到按值传递的效果：

```
//: C03:PassByValue.cpp
#include <iostream>
using namespace std;

void f(int a) {
    cout << "a = " << a << endl;
    a = 5;
    cout << "a = " << a << endl;
}

int main() {
    int x = 47;
    cout << "x = " << x << endl;
    f(x);
    cout << "x = " << x << endl;
} ///:~
```



在函数`f()`中，`a`是一个局部变量 (*local variable*)，它只有在调用函数`f()`期间存在。因为它是一个函数参数，所以调用函数时通过参数传递来初始化`a`的值；在`main()`中参数是`x`，其值为47，所以当调用函数`f()`时，这个值被拷贝到`a`中。

当运行这个程序时，我们会看到：

```
x = 47
a = 47
a = 5
x = 47
```

当然，最初，`x`的值是47。调用`f()`时，在函数调用期间为变量`a`分配临时空间，拷贝`x`的值给`a`来初始化它，这可以通过打印结果得到验证。当然，我们可以改变`a`的值并显示它被改变。但是`f()`调用结束时，分配给`a`的临时空间就消失了，我们可以看到，在`a`和`x`之间的曾经发生过的惟一联系，是在把`x`的值拷贝到`a`的时候。

当在函数`f()`内部时，变量`x`就是外部对象 (*outside object*) (我用的术语)。显然，改变局部变量并不会影响外部变量，因为它们分别放在存储空间的不同位置。但是，如果我们的确想修改外部对象那又该怎么办呢？这时指针就该派上用场了。在某种意义上，指针是另一个变量的别名。所以如果我们不是传递一个普通的值而是传递一个指针给函数，实际上就是传递外部对象的别名，使函数能修改外部对象，例如：

```
//: C03:PassAddress.cpp
#include <iostream>
using namespace std;

void f(int* p) {
    cout << "p = " << p << endl;
    cout << "*p = " << *p << endl;
    *p = 5;
    cout << "p = " << p << endl;
}

int main() {
    int x = 47;
    cout << "x = " << x << endl;
    cout << "&x = " << &x << endl;
    f(&x);
    cout << "x = " << x << endl;
} ///:~
```

现在函数`f()`把指针作为参数，并且在赋值期间间接引用这个指针，这就使得外部对象`x`被修改。这时的输出是：

```
x = 47
&x = 0065FE00
p = 0065FE00
*p = 47
p = 0065FE00
x = 5
```

注意，`p`中的值就是变量`x`的地址，指针`p`的确是指向变量`x`。如果这还不够令人信服，当改变指针`p`指向的变量值并间接引用赋值为5，我们看到变量`x`的值现在已经改变为5了。

因此，通过给函数传递指针可以允许函数修改外部对象。后面我们将看到指针有很多其

他的用途，但是这是最基本的，可能也是最常用的用途。

### 3.4.6 C++引用简介

在C和C++中指针的作用基本上是一样的，但是C++增加了另外一种给函数传递地址的途径。这就是按引用传递 (*pass-by-reference*)，它也存在一些其他的编程语言中，并不是C++的发明。

可能一开始我们会觉得没有必要使用引用，可以不用引用编写所有的程序。一般说来，除开在本书后面将要知道的一些重要地方，这是确实的。在后面我们将对引用有更多的了解，但是基本思想和前面所述的指针的使用是一样的：我们可以用引用传递参数地址。引用和指针的不同之处在于，带引用的函数调用比带指针的函数调用在语法构成上更清晰（在某种情况下，使用引用实质上的确只是语法构成上不同）。如果使用引用来修改程序 **PassAddress.cpp**，我们能看到在 **main()** 中函数调用的不同：

```
//: C03:PassReference.cpp
#include <iostream>
using namespace std;

void f(int& r) {
    cout << "r = " << r << endl;
    cout << "&r = " << &r << endl;
    r = 5;
    cout << "r = " << r << endl;
}

int main() {
    int x = 47;
    cout << "x = " << x << endl;
    cout << "&x = " << &x << endl;
    f(x); // Looks like pass-by-value,
        // is actually pass by reference
    cout << "x = " << x << endl;
} ///:~
```

在函数 **f()** 的参数列表中，不用 **int\*** 来传递指针，而是用 **int&** 来传递引用。在 **f()** 中，如果仅仅写 '**r**'（如果 **r** 是一个指针，会产生一个地址值）会得到 **r** 引用的变量值。如果对 **r** 赋值，实际上是给 **r** 引用的变量赋值。事实上，得到 **r** 中存放的地址值的惟一方法是用 '**&**' 运算符。

在函数 **main()** 中，我们能看到引用在调用函数 **f()** 中的重要作用，其语法形式还是 **f(x)**。尽管这看起来像是一般的按值传递，但是实际上引用的作用是传递地址，而不是值的一个拷贝。输出结果是：

```
x = 47
&x = 0065FE00
r = 47
&r = 0065FE00
r = 5
x = 5
```

所以我们可以看到，以引用传递允许一个函数去修改外部对象，就像传递一个指针所做的那样（读者可能也注意到引用使得地址传递这个事实不太明显，这在本书的后面会得到检验）。因此，通过这个简单的介绍，我们可以认为引用仅仅是语法上的一种不同方法（有时称

为“语法糖”),它和指针完成同样的任务:允许函数去改变外部对象。

### 3.4.7 用指针和引用作为修饰符

迄今为止,我们已经看到了基本的数据类型`char`、`int`、`float`和`double`,看到了修饰符`signed`、`unsigned`、`short`和`long`,它们可以和基本的数据类型结合使用。现在我们增加了指针和引用(它们与基本数据类型和修饰符是独立的),所以可能产生三倍的结合:

```

//: C03:AllDefinitions.cpp
// All possible combinations of basic data types,
// specifiers, pointers and references
#include <iostream>
using namespace std;

void f1(char c, int i, float f, double d);
void f2(short int si, long int li, long double ld);
void f3(unsigned char uc, unsigned int ui,
    unsigned short int usi, unsigned long int uli);
void f4(char* cp, int* ip, float* fp, double* dp);
void f5(short int* sip, long int* lip,
    long double* ldp);
void f6(unsigned char* ucp, unsigned int* uip,
    unsigned short int* usip,
    unsigned long int* ulip);
void f7(char& cr, int& ir, float& fr, double& dr);
void f8(short int& sir, long int& lir,
    long double& ldr);
void f9(unsigned char& ucr, unsigned int& uir,
    unsigned short int& usir,
    unsigned long int& ulir);

int main() {} ///:~

```

当传递对象进出函数时,指针和引用也能工作;我们将会在后面的一章了解到这些内容。

这里有和指针一起工作的另一种类型:`void`。如果声明指针是`void*`,它意味着任何类型的地址都可以间接引用那个指针(而如果声明`int*`,则只能对`int`型变量的地址间接引用那个指针)。例如:

```

//: C03:VoidPointer.cpp
int main() {
    void* vp;
    char c;
    int i;
    float f;
    double d;
    // The address of ANY type can be
    // assigned to a void pointer:
    vp = &c;
    vp = &i;
    vp = &f;
    vp = &d;
} ///:~

```

一旦我们间接引用一个`void*`,就会丢失关于类型的信息。这意味着在使用前,必须转换

为正确的类型:

```
//: C03:CastFromVoidPointer.cpp
int main() {
    int i = 99;
    void* vp = &i;
    // Can't dereference a void pointer:
    // *vp = 3; // Compile-time error
    // Must cast back to int before dereferencing:
    *((int*)vp) = 3;
} ///:~
```

转换(int\*)vp告诉编译器把void\*当做int\*处理, 因此可以成功地对它间接引用。读者可能注意到, 这个语法很难看, 的确如此, 但是更糟的是, void\*在语言类型系统中引入了一个漏洞。也就是说, 它允许甚至是提倡把一种类型看做另一种类型。在上面的例子中, 通过把vp转换为int\*, 把一个整型看做是一个整型, 但是, 并没有说不能把它转换为一个char\*或double\*, 这将改变已经分配给int的存储空间的大小, 可能会引起程序崩溃。一般来说, 应当避免使用void指针, 只有在一些少见的特殊情况下才用, 到本书的后面才需要考虑这些。

我们不能使用void引用, 其原因将在第11章说明。

### 3.5 作用域

作用域规则告诉我们一个变量的有效范围, 它在哪里创建, 在哪里销毁(也就是说, 超出了作用域)。变量的有效作用域从它的定义点开始, 到和定义变量之前最邻近的开括号配对的第一个闭括号。也就是说, 作用域由变量所在的最近一对括号确定。说明如下:

```
//: C03:Scope.cpp
// How variables are scoped
int main() {
    int scp1;
    // scp1 visible here
    {
        // scp1 still visible here
        //.....
        int scp2;
        // scp2 visible here
        //.....
        {
            // scp1 & scp2 still visible here
            //..
            int scp3;
            // scp1, scp2 & scp3 visible here
            // ...
        } // <-- scp3 destroyed here
        // scp3 not available here
        // scp1 & scp2 still visible here
        // ...
    } // <-- scp2 destroyed here
    // scp3 & scp2 not available here
    // scp1 still visible here
    //..
} // <-- scp1 destroyed here
///:~
```



上面的例子表明什么时候变量是可见的，什么时候变量是不可用的（即变量超出其作用域）。只有在变量的作用域内，才能使用它。作用域可以嵌套，即在一对大括号里面有其他的大括号对。嵌套意味着可以在我们所处的作用域内访问外层作用域的一个变量。上面的例子中，变量**scp1**在所有的作用域内都可用，而**scp3**只能在最里面的作用域内才可用。

### 3.5.1 实时定义变量

正如在本章前面提到的那样，定义变量时，C和C++有着显著的区别。这两种语言都要求变量使用前必须定义，但是C（和很多其他的传统过程语言）强制在作用域的开始处就定义所有的变量，以便在编译器创建一个块时，能给所有这些变量分配空间。

读C代码时，进入一个作用域，首先看到的是一个变量的定义块。在块的开始部分声明所有的变量，要求程序员以一种特定的方式写程序，因为语言的实现细节需要这样。大多数人在写代码之前并不知道他们将要使用的所有变量，所以他们必须不停地跳转回块的开头来插入新的变量，这是很不方便的，也会引起错误。这些变量定义对读者来说并没有很多含义，它们实际上只是容易引起混乱，因为它们出现的地方远离使用它们的上下文。

C++(不是C)允许在作用域内的任意地方定义变量，所以可以在正好使用它之前定义。此外，可以在定义变量时对它进行初始化以防止犯某种类型的错误。以这种方式定义变量使得编写代码更容易，减少了在一个作用域内不停地来回跳转造成的问题。因为可以在使用变量的上下文中看到所定义的变量，所以代码更容易理解。同时定义并初始化一个变量是非常重要的。通过使用变量的方式我们可以看到初始化一个变量值的意义。

我们还可以在**for**循环和**while**循环的控制表达式内定义变量，在**if**语句的条件表达式和**switch**的选择器语句内定义变量。下面是一个显示随时定义变量的例子：

```

//: C03:OnTheFly.cpp
// On-the-fly variable definitions
#include <iostream>
using namespace std;
int main() {
    //..
    { // Begin a new scope
        int q = 0; // C requires definitions here
        //..
        // Define at point of use:
        for(int i = 0; i < 100; i++) {
            q++; // q comes from a larger scope
            // Definition at the end of the scope:
            int p = 12;
        }
        int p = 1; // A different p
    } // End scope containing q & outer p
    cout << "Type characters:" << endl;
    while(char c = cin.get() != 'q') {
        cout << c << " wasn't it" << endl;
        if(char x = c == 'a' || c == 'b')
            cout << "You typed a or b" << endl;
        else
            cout << "You typed " << x << endl;
    }
}

```





```

cout << "Type A, B, or C" << endl;
switch(int i = cin.get()) {
    case 'A': cout << "Snap" << endl; break;
    case 'B': cout << "Crackle" << endl; break;
    case 'C': cout << "Pop" << endl; break;
    default: cout << "Not A, B or C!" << endl;
}
} ///:~

```

在最内层的作用域里，**p**是在作用域结束之前定义的，所以它只是一个毫无意义的表示（但它表明可以在任何地方定义一个变量）。在外层作用域中的**p**也是一样的情况。

在**for**循环的控制表达式中**i**的定义正是一个在需要的地方定义变量的例子（只能在C++中这样做）。**i**的作用域是**for**循环控制的表达式的作用域，所以可以轮到下一次**for**循环并重新使用**i**。这是在C++中一个非常方便和常用的用法；**i**是循环计数器的一个常用名字，我们不必费神取新的名字。

尽管例子表明在**while**语句、**if**语句和**switch**语句中也可以定义变量，但是可能因为语法受到许多限制，这种定义不如在**for**的表达式中常用。例如，我们不能有任何插入括号。也就是说，不可以写出：

```
while((char c = cin.get()) != 'q')
```

附加的括号似乎是合理的，并且能做很有用的事，但因为无法使用它们，结果就不像所希望的那样。问题是因为‘!=’比‘=’的优先级高，所以**char c**最终含有的值是由**bool**转换为**char**的。当打印出来时，我们在很多终端上会看到一个笑脸字符。

通常，可以认为在**while**语句、**if**语句和**switch**语句中定义变量的能力是为了完备性，但是惟一使用这种变量定义的地方可能是在**for**循环中（在那里可能使用得十分频繁）。

### 3.6 指定存储空间分配

创建一个变量时，我们拥有指定变量生存期的很多选择，指定怎样给变量分配存储空间，以及指定编译器怎样处理这些变量。

#### 3.6.1 全局变量

全局变量是在所有函数体的外部定义的，程序的所有部分（甚至其他文件中的代码）都可以使用。全局变量不受作用域的影响，总是可用的（也就是说，全局变量的生命期一直到程序的结束）。如果在一个文件中使用**extern**关键字来声明另一个文件中存在的全局变量，那么这个文件可以使用这个数据。例如：

```

//: C03:Global.cpp
//{L} Global2
// Demonstration of global variables
#include <iostream>
using namespace std;

int globe;
void func();
int main() {
    globe = 12;
}

```

```

    cout << globe << endl;
    func(); // Modifies globe
    cout << globe << endl;
} ///:~

```

下面的程序把**globe**作为一个外部变量来访问：

```

//: C03:Global2.cpp {O}
// Accessing external global variables
extern int globe;
// (The linker resolves the reference)
void func() {
    globe = 47;
} ///:~

```

变量**globe**的存储空间是由程序**Global.cpp**中的定义创建的，在**Global2.cpp**的代码中可以访问同一个变量。由于**Global2.cpp**和**Global.cpp**的代码是分段编译的，必须通过声明：

```
extern int globe;
```

告诉编译器变量存在哪里。

运行这个程序时，会看到函数**func()**的调用的确影响**globe**的全局实例。

在**Global.cpp**中，可能看到下面这个特殊的注释标记（这是我自己的设计）：

```
//[L} Global2
```

这是说要创建最后的程序，带有**Global2**名字的目标文件必须被连接进来（这里没有扩展名是因为目标文件的扩展名在不同的系统中是不一样的）。在**Global2.cpp**中，第一行有另一个特殊的注释标记**{O}**，意思是“不要从这个文件生成可执行文件，编译它是为了把它连接进一些其他的可执行文件中。”本书第2卷中的**ExtractCode.cpp**程序（在[www.BruceEckel.com](http://www.BruceEckel.com)上可以下载）阅读这些标记并生成适当的**makefile**使得每一个文件被正确地编译（在本章结束时将会了解**makefile**）。

### 3.6.2 局部变量

局部变量出现在一个作用域内，它们是局限于一个函数的。局部变量经常被称为自动变量（*automatic variable*），因为它们在进入作用域时自动生成，离开作用域时自动消失。关键字**auto**可以显式地说明这个问题，但是局部变量默认为**auto**，所以没有必要声明为**auto**。

#### 3.6.2.1 寄存器变量

寄存器变量是一种局部变量。关键字**register**告诉编译器“尽可能快地访问这个变量”。加快访问速度取决于实现，但是，正如名字所暗示的那样，这经常是通过在寄存器中放置变量来做到的。这并不能保证将变量放置在寄存器中，甚至也不能保证提高访问速度。这只是对编译器的一个暗示。

使用**register**变量是有限制的。不可能得到或计算**register**变量的地址。**register**变量只能在一个块中声明（不可能有全局的或静态的**register**变量）。然而可以在一个函数中（即在参数表中）使用**register**变量作为一个形式参数。

一般地，不应当推测编译器的优化器，因为它可能比我们做得更好。因此，最好避免使用关键字**register**。

### 3.6.3 静态变量

关键字**static**有一些独特的意义。通常，函数中定义的局部变量在函数作用域结束时消失。当再次调用这个函数时，会重新创建该变量的存储空间，其值会被重新初始化。如果想使局部变量的值在程序的整个生命期里仍然存在，我们可以定义函数的局部变量为**static**（静态的），并给它一个初始值。初始化只在函数第一次调用时执行，函数调用之间变量的值保持不变。用这种方式，函数可以“记住”函数调用之间的一些信息片断。

我们可能奇怪为什么不使用全局变量。**static**变量的优点是在函数范围之外它是不可用的，所以它不可能被轻易地改变。这会使错误局部化。

下面是一个使用**static**变量的例子：

```
//: C03:Static.cpp
// Using a static variable in a function
#include <iostream>
using namespace std;

void func() {
    static int i = 0;
    cout << "i = " << ++i << endl;
}

int main() {
    for(int x = 0; x < 10; x++)
        func();
} ///:~
```

每一次在for循环中调用函数**func()**时，它都打印不同的值。如果不使用关键字**static**，打印出的值总是‘1’。

**static**的第二层意思和前面的含义相关，即“在某个作用域外不可访问”。当应用**static**于函数名和所有函数外部的变量时，它的意思是“在文件的外部不可以使用这个名字”。函数名或变量是局部于文件的；我们说它具有文件作用域（*file scope*）。例如，编译和连接下面两个文件会引起连接器错误：

```
//: C03:FileStatic.cpp
// File scope demonstration. Compiling and
// linking this file with FileStatic2.cpp
// will cause a linker error

// File scope means only available in this file:
static int fs;

int main() {
    fs = 1;
} ///:~
```

尽管在下面的文件中变量**fs**被声明为**extern**，但是连接器不会找到它，因为在**FileStatic.cpp**中它被声明为**static**。

```
//: C03:FileStatic2.cpp {0}
// Trying to reference fs
extern int fs;
```

```
void func() {
    fs = 100;
} ///:~
```

**static**说明符也可能在一个类中使用。当在本书的后面了解了如何创建类的时候，再对此作出解释。

### 3.6.4 外部变量

前面已经简要地描述和说明了**extern**关键字。它告诉编译器存在着一个变量和函数，即使编译器在当前编译的文件中没有看到它。这个变量或函数可能在另一个文件中或者在当前文件的后面定义。下面是一个例子：

```
///: C03:Forward.cpp
// Forward function & data declarations
#include <iostream>
using namespace std;

// This is not actually external, but the
// compiler must be told it exists somewhere:
extern int i;
extern void func();

int main() {
    i = 0;
    func();
}
int i; // The data definition
void func() {
    i++;
    cout << i;
} ///:~
```

当编译器遇到‘**extern int i**’时，它知道*i*肯定作为全局变量存在于某处。当编译器看到变量*i*的定义时，并没有看到别的声明，所以知道它在文件的前面已经找到了同样声明的*i*。如果已经把变量*i*定义为**static**，又要告诉编译器，*i*是全局定义的（通过**extern**），但是，它也有文件作用域（通过**static**），所以编译器会产生错误。

#### 3.6.4.1 连接

为了理解C和C++程序的行为，必须对连接（*linkage*）有所了解。在一个执行程序中，标识符代表存放变量或被编译过的函数体的存储空间。连接用连接器所见的方式描述存储空间。连接方式有两种：内部连接（*internal linkage*）和外部连接（*external linkage*）。

内部连接意味着只对正被编译的文件创建存储空间。用内部连接，别的文件可以使用相同的标识符或全局变量，连接器不会发现冲突——也就是为每一个标识符创建单独的存储空间。在C和C++中，内部连接是由关键字**static**指定的。

外部连接意味着为所有被编译过的文件创建一片单独的存储空间。一旦创建存储空间，连接器必须解决所有对这片存储空间的引用。全局变量和函数名有外部连接。通过用关键字**extern**声明，可以从其他文件访问这些变量和函数。函数之外定义的所有变量（在C++中除了**const**）和函数定义默认为外部连接。可以使用关键字**static**特地强制它们具有内部连接，也可以在定义时使用关键字**extern**显式指定标识符具有外部连接。在C中，不必用**extern**定义变量

或函数，但是在C++中对于**const**有时必须使用。

调用函数时，自动（局部）变量只是临时存在于堆栈中。连接器不知道自动变量，所以这些变量没有连接。

### 3.6.5 常量

在旧版本（标准前）的C中，如果想建立一个常量，必须使用预处理器：

```
#define PI 3.14159
```

无论在何地使用**PI**，都会被预处理器用值3.14159代替（在C和C++中都可以使用这个方法）。

当使用预处理器创建常量时，我们在编译器的范围之外能控制这些常量。对名字**PI**上不进行类型检查，也不能得到**PI**的地址（所以不能向**PI**传递一个指针和一个引用）。**PI**不能是用户定义的类型变量。**PI**的意义是从定义它的地方持续到文件结束的地方；预处理器并不识别作用域。

C++引入了命名常量的概念，命名常量就像变量一样，只是它的值不能改变。修饰符**const**告诉编译器这个名字表示常量。不管是内部的还是用户定义的数据类型都可以定义为**const**。如果定义了某对象为常量，然后试图修改它，编译器将会产生错误。

必须用下述方式说明一个常量类型：

```
const int x = 10;
```

在标准C和C++中，可以在参数列表中使用命名常量，即使列表中的参数是指针或引用（也就是说，可以获得**const**的地址）。**const**就像正常的变量一样有作用域，所以可以在函数中“隐藏”一个**const**，确保名字不会影响程序的其余部分。

**const**由C++采用，并加进标准C中，尽管它们很不一样。在C中，编译器对待**const**如同变量一样，只不过带有一个特殊的标记，意思是“不要改变我”。当在C中定义**const**时，编译器为它创建存储空间，所以如果在两个不同的文件中（或在头文件中）定义多个同名的**const**，连接器将生成发生冲突的错误消息。在C中使用**const**和在C++中使用**const**是完全不一样的（简而言之，在C++中使用得更好）。

#### 3.6.5.1 常量值

在C++中，一个**const**必须有初始值（在C中不是这样）。内建类型的常量值可以表示为十进制、八进制、十六进制、浮点数（不幸的是，二进制数被认为是不重要的）或字符。

如果没有其他的线索，编译器会认为常量值是十进制。数值47、0和1101都被认为是十进制数。

常量值前带0被认为是八进制数（基数为8）。基数为8的数值只能含有数字0~7，编译器标记其他数字为错误。017是一个合法的八进制数（相当于基数为10的数值15）。

常量值前带0x被认为是十六进制数（基数为16）。基数为16的数值只能含有数字0~9和字母a~f或A~F。0x1fe是一个合法十六进制数（相当于基数为10的数值510）。

浮点数可以含有小数点和指数幂（用e表示，意思是“10的幂”）。小数点和e都可以任选。如果给一个浮点变量赋一个常量值，编译器会取得这个常量值并把它转换为浮点数（这个过程是隐式类型转换（*implicit type conversion*）的一种形式）。但是，使用小数点或e对于提醒

读者当前正在使用的是浮点数是一个好主意；一些更旧的编译器也会需要这种暗示。

合法的浮点常量值包括：1e4、1.0001、47.0、0.0和-1.159e-77。我们可以对数加后缀强加浮点数类型：**f**或**F**强加**float**型，**L**或**l**强加**long double**型，否则是**double**型。

字符常量是用单引号括起来的字符，如‘A’、‘0’、‘ ’。注意字符‘0’（ASCII 96）和数值0之间存在巨大差别。用“反斜线”表示一些特殊的字符：‘\n’（换行），‘\t’（制表符），‘\\’（反斜线），‘\r’（回车），‘\’（双引号），‘\’（单引号），等等。也可以用八进制表示字符常量（如‘\17’）或用十六进制表示字符常量（如‘\xff’）。

### 3.6.6 volatile变量

限定词**const**告诉编译器“这是不会改变的”（这就允许编译器执行额外的优化）；而限定词**volatile**则告诉编译器“不知道何时会改变”，防止编译器依据变量的稳定性作任何优化。当读在代码控制之外的某个值时，例如读一块通信硬件中的寄存器，将使用这个关键字。无论何时需要**volatile**变量的值，都能读到，即使在该行之前刚刚读过。

“在代码的控制之外”的某个存储空间的一个特殊例子是在多线程程序中。如果正在观察被另一个线程或进程修改的特殊标识符，这个标识符应该是**volatile**的，所以编译器不会认为它能够对标识符的多次读入进行优化。

注意当编译器不进行优化时，**volatile**可能不起作用，但是当开始优化代码时（当编译器开始寻找冗余的读入时），可以防止出现重大的错误。

后面有一章将进一步阐述**const**和**volatile**关键字。

## 3.7 运算符及其使用

本节说明C和C++中的所有运算符。

所有的运算符都会从它们的操作数中产生一个值。除了赋值、自增、自减运算符之外，运算符所产生的值不会修改操作数。修改操作数被称为副作用(*side effect*)。一般使用修改操作数的运算符就是为了产生这种副作用，但是应该记住它们所产生的值就像没有副作用的运算符产生的值一样都是可以使用的。

### 3.7.1 赋值

赋值操作由运算符“=”实现。这意味着“取右边的值[通常称之为右值(*rvalue*)]并把它拷贝给左边[通常称之为左值(*lvalue*)]”。右值可以是任意的常量、变量或能产生值的表达式，但是左值必须是一个明确命名的变量（也就是说，应该有一个存储数据的物理空间）。例如，可以给一个变量赋值常量(**A = 4;**)，但是不能给常量赋任何值，因为它不能是左值（不能用**4 = A;**）。

### 3.7.2 数学运算符

基本的数学运算符和在大多数的编程语言中使用的一样：加(+)、减(-)、除(/)、乘(\*)和取模(%)；从整数相除得到余数)。整数相除会截取结果的整数部分（不舍入）。浮点数不能使用取模运算符。

C和C++也使用一种简化的符号来同时执行操作和赋值。这是由一个运算符后面跟着一个

等号来表示的，并且与语言中的各种运算符结合（只要有意义）。例如，要给变量`x`加4并赋值给`x`作为结果，可以写成`x += 4;`。

下面例子显示了数学运算符的使用：

```
//: C03:Mathops.cpp
// Mathematical operators
#include <iostream>
using namespace std;

// A macro to display a string and a value.
#define PRINT(STR, VAR) \
    cout << STR " = " << VAR << endl

int main() {
    int i, j, k;
    float u, v, w; // Applies to doubles, too
    cout << "enter an integer: ";
    cin >> j;
    cout << "enter another integer: ";
    cin >> k;
    PRINT("j",j); PRINT("k",k);
    i = j + k; PRINT("j + k",i);
    i = j - k; PRINT("j - k",i);
    i = k / j; PRINT("k / j",i);
    i = k * j; PRINT("k * j",i);
    i = k % j; PRINT("k % j",i);
    // The following only works with integers:
    j %= k; PRINT("j %= k", j);
    cout << "Enter a floating-point number: ";
    cin >> v;
    cout << "Enter another floating-point number:";
    cin >> w;
    PRINT("v",v); PRINT("w",w);
    u = v + w; PRINT("v + w", u);
    u = v - w; PRINT("v - w", u);
    u = v * w; PRINT("v * w", u);
    u = v / w; PRINT("v / w", u);
    // The following works for ints, chars,
    // and doubles too:
    PRINT("u", u); PRINT("v", v);
    u += v; PRINT("u += v", u);
    u -= v; PRINT("u -= v", u);
    u *= v; PRINT("u *= v", u);
    u /= v; PRINT("u /= v", u);
} //:~
```

当然所有赋值的右值都可以更为复杂。

### 3.7.2.1 预处理宏介绍

注意，使用宏**PRINT**( )可以节省输入（和避免输入错误！）。传统上用大写字母来命名预处理宏以便突出它——后面我们很快会了解到宏有可能会变得危险（它们也可能非常有用）。

跟在宏名后面的括号中的参数会被闭括号后面的所有代码替代。只要在调用宏的地方，预处理程序就删除名字**PRINT**并替换代码，所以使用宏名时编译器不会报告任何错误信息，它并不对参数做任何类型检查（正如本章后面宏调试中显示的那样，后者可能是有益的）。



### 3.7.3 关系运算符

关系运算符在操作数之间建立一种关系。如果关系为真，则产生布尔（在C++中用关键字**bool**表示）值**true**；如果关系为假，则产生布尔值**false**。关系运算符有：小于(<), 大于(>), 小于等于(<=), 大于等于(>=), 等于(==), 不等于(!=)。在C和C++中，它们可以使用所有的内建数据类型。在C++中，对用户所定义的数据类型可以给出它们的特殊定义（在第12章讨论运算符重载时将了解这些内容）。

### 3.7.4 逻辑运算符

逻辑运算符“与”(&&)和“或”(||)依据它们的参数的逻辑关系产生**true**或**false**。记住在C和C++中，如果语句是非零值则为**true**，如果是零则为**false**。如果打印一个**bool**值，一般会看到‘1’表示**true**、‘0’表示**false**。

下面例子使用了关系运算符和逻辑运算符：

```
//: C03:Boolean.cpp
// Relational and logical operators.
#include <iostream>
using namespace std;
int main() {
    int i, j;
    cout << "Enter an integer: ";
    cin >> i;
    cout << "Enter another integer: ";
    cin >> j;
    cout << "i > j is " << (i > j) << endl;
    cout << "i < j is " << (i < j) << endl;
    cout << "i >= j is " << (i >= j) << endl;
    cout << "i <= j is " << (i <= j) << endl;
    cout << "i == j is " << (i == j) << endl;
    cout << "i != j is " << (i != j) << endl;
    cout << "i && j is " << (i && j) << endl;
    cout << "i || j is " << (i || j) << endl;
    cout << " (i < 10) && (j < 10) is "
        << ((i < 10) && (j < 10)) << endl;
} ///:~
```

在上面的程序中，我们可以用**float**或**double**代替**int**定义。但是，注意浮点数和零的比较是很严格的，一个数和另一个数即使只有最小小数位不同仍然是“不相等”。一个最小小数位大于0的浮点数仍为真。

### 3.7.5 位运算符

位运算符允许在一个数中处理个别的位（因为浮点数使用一种特殊的内部格式，所以位运算符只适用于整型**char**、**int**和**long**）。位运算符对参数中的相应位做布尔代数运算来产生结果。

如果两个输入位都是1，则“与”运算符(&)在结果位上产生1，否则为0。如果两个输入位有一个是1，则“或”运算符(|)在结果位上产生1，只有当两个输入位都是0时，结果位才为0。如果两个输入位之一是1而不是同时为1，则位的异或运算符**xor**(^)的结果位为1。位的“非”运算符(~, 也称为补运算符)是一个一元运算符，它只带一个参数（其他的运算

符都是二元运算符)。非运算符运算的结果和输入位相反，即输入位为0时结果位为1，输入位为1时结果位为0。

位运算符可以和“=”结合来统一运算和赋值：**&=**、**|=**和**^=**都是合法运算（因为~是一元运算符，所以不能和=结合）。

### 3.7.6 移位运算符

移位运算符也是对位的操纵。左移位运算符(**<<**)引起运算符左边的操作数向左移动，移动位数由运算符后面的操作数指定。右移位运算符(**>>**)引起运算符左边的操作数向右移动，移动位数由运算符后面的操作数指定。如果移位运算符后面的值比运算符左边的操作数的位数大，则结果是不定的。如果左边的操作数是无符号的，右移是逻辑移位，所以最高位补零。如果左边的操作数是有符号的，右移可能是也可能不是逻辑移位（也就是说，行为是不定的）。

移位可以和等号结合(**<<=**或**>>=**)。左值由左值按右值移位后的结果代替。

下面是一个例子，说明所有涉及位运算的运算符的使用。首先，这里单独创建了一个通用的函数，用二进制格式打印一个字节，所以这个函数很容易被重用。头文件声明了这个函数：

```
//: C03:printBinary.h
// Display a byte in binary
void printBinary(const unsigned char val);
///:~
```

下面是这个函数的实现：

```
//: C03:printBinary.cpp {0}
#include <iostream>
void printBinary(const unsigned char val) {
    for(int i = 7; i >= 0; i--)
        if(val & (1 << i))
            std::cout << "1";
        else
            std::cout << "0";
} ///:~
```

函数**printBinary()**取出一个字节并一位一位地显示出来。表达式

```
(1 << i)
```

在每一个相继位的位置产生一个1，例如：00000001, 00000010, 等等。如果这一位和变量**val**按位与并且结果不是零，就表明**val**的这一位为1。

最后，在例子中使用下面的函数显示位操作运算符：

```
//: C03:Bitwise.cpp
//{L} printBinary
// Demonstration of bit manipulation
#include "printBinary.h"
#include <iostream>
using namespace std;

// A macro to save typing:
#define PR(STR, EXPR) \
    cout << STR; printBinary(EXPR); cout << endl;
```



```

int main() {
    unsigned int getval;
    unsigned char a, b;
    cout << "Enter a number between 0 and 255: ";
    cin >> getval; a = getval;
    PR("a in binary: ", a);
    cout << "Enter a number between 0 and 255: ";
    cin >> getval; b = getval;
    PR("b in binary: ", b);
    PR("a | b = ", a | b);
    PR("a & b = ", a & b);
    PR("a ^ b = ", a ^ b);
    PR("~a = ", ~a);
    PR("~b = ", ~b);
    // An interesting bit pattern:
    unsigned char c = 0x5A;
    PR("c in binary: ", c);
    a |= c;
    PR("a |= c; a = ", a);
    b &= c;
    PR("b &= c; b = ", b);
    b ^= a;
    PR("b ^= a; b = ", b);
} ///:~

```

再一次使用预处理宏节省输入。它打印你选择的字符串，然后是一个表达式的二进制表示形式，再后是换行。

在`main()`中，变量都是**unsigned**的。这是因为一般来说，在使用字节进行工作时并不希望用带符号数。对于变量`getval`而言，可能要使用**int**来替代**char**，因为语句“`cin >>`”以另一种方式把第一个数字看做是一个字符。通过把`getval`赋值给**a**和**b**，该值被转换为一个单独的字节（通过对它截尾）。

“`<<`”和“`>>`”实现位的移位功能，但是当移位越出数的一端时，那些位就会丢失（这就是通常所说的，那些位掉进了神秘的位桶（*bit bucket*）中，丢弃在这个桶中的位有可能需要重用）。操作位的时候，也可以执行旋转（*rotation*），即在一端移掉的位插入到另一端，好像它们在绕着一个回路旋转。尽管大多数计算机处理器提供了机器级的旋转命令（所以我们会在这种处理器的汇编语言中看到它），但在C和C++中，不直接支持旋转。大概C的设计者认为对“旋转”的处理应该适可而止（正如他们说的那样，他们的目标是建立最小的语言），因此我们可以建立自己的旋转命令。例如下面是实现左旋和右旋的函数：

```

//: C03:Rotation.cpp {0}
// Perform left and right rotations

unsigned char rol(unsigned char val) {
    int highbit;
    if(val & 0x80) // 0x80 is the high bit only
        highbit = 1;
    else
        highbit = 0;
    // Left shift (bottom bit becomes 0):
    val <<= 1;

```

```

    // Rotate the high bit onto the bottom:
    val |= highbit;
    return val;
}

unsigned char ror(unsigned char val) {
    int lowbit;
    if(val & 1) // Check the low bit
        lowbit = 1;
    else
        lowbit = 0;
    val >>= 1; // Right shift by one position
    // Rotate the low bit onto the top:
    val |= (lowbit << 7);
    return val;
} ///:~

```

试着在程序**Bitwise.cpp**中使用这些函数。注意，在使用这些函数前编译器必须在**Bitwise.cpp**中看到**rol()**和**ror()**的定义（或者至少是声明）。

通常情况下，使用位函数的效率非常高，因为它们被直接翻译成汇编语言语句。有时一个单独的C或C++语句会产生一行单独的汇编代码。

### 3.7.7 一元运算符

位的非运算不是惟一使用一个参数的运算符。和它一样，逻辑非(!)对一个**true**值得到一个**false**值。一元减(-)和一元加(+)是和二元减和二元加一样的运算符；根据表达式的书写方式，编译器能辨别属于哪一种用法。例如，语句

```
x = -a;
```

有明确的含义。

编译器可以理解

```
x = a * -b;
```

但是读者可能迷惑，所以写成

```
x = a * (-b);
```

更保险。

一元减得到一个负值。一元加实际上并不做任何事，只是和一元减相对应。

本章前面介绍了增量和减量运算符(++和--)。它们是涉及赋值的运算符中仅有的有副作用的运算符。这两个运算符使变量增加或减少一个单位，尽管对于不同的数据类型，“单位”可能有不同的含义——特别是对指针来说。

最后的一元运算符有C和C++中的地址运算符(&)，间接引用(\*和->)和强制类型转换运算符，以及C++中的**new**和**delete**。在本章的叙述中，地址和间接引用只与指针一起使用。类型转换在本章后面叙述，**new**和**delete**将在第4章介绍。

### 3.7.8 三元运算符

三元运算符**if-else**与众不同，因为它有三个操作数。这的确是一个运算符因为它产生一个值，而不是像一般的**if-else**语句那样。它由三个表达式组成：如果第一个表达式（后面跟有一

个问号?)的计值为**true**,则对紧跟在问号后面的表达式求值,它的结果就是运算符的结果。如果第一个表达式为**false**,就执行第三个表达式(在冒号后面),它的结果就是运算符的结果。

可以使用**if-else**这个条件运算符的副作用或者它产生的值。下面的代码段说明了这两种情况:

```
a = --b ? b : (b = -99);
```

这里,条件产生右值。如果**b**自减运算的结果非零,则把**b**的值赋给**a**。如果**b**变为零,**a**和**b**都被赋值为-99。**b**总是在递减,但是只有在**b**递减为0时,它才会被赋值为-99。可以使用如下不带“**a =**”的类似语句来利用它的副作用:

```
--b ? b : (b = -99);
```

在这里第二个**b**是多余的,因为运算符产生的值是无用的。但在“?”和“:”之间需要一个表达式。在这种情况下,这个表达式可以是一个常量,它能使代码运行得更快一点。

### 3.7.9 逗号运算符

逗号并不只是在定义多个变量时用来分隔变量,例如:

```
int i, j, k;
```

当然,它也用于函数参数列表中。然而,它也可能作为一个运算符用于分隔表达式。在这种情况下,它只产生最后一个表达式的值。在逗号分隔的列表中,其余的表达式只完成它们的副作用。下面的例子自增一串变量,并把最后一个作为右值:

```
//: C03:CommaOperator.cpp
#include <iostream>
using namespace std;
int main() {
    int a = 0, b = 1, c = 2, d = 3, e = 4;
    a = (b++, c++, d++, e++);
    cout << "a = " << a << endl;
    // The parentheses are critical here. Without
    // them, the statement will evaluate to:
    (a = b++), c++, d++, e++;
    cout << "a = " << a << endl;
} ///:~
```

通常,除了作为一个分隔符,逗号最好不作他用,因为人们不习惯把它看做是运算符。

### 3.7.10 使用运算符时的常见问题

如上所述,使用运算符时的一个问题是总不愿使用括号,即使在还不确定一个表达式如何计算时(可以查阅当前的C手册中表达式的计算顺序)。

另一个十分常见的错误如下所示:

```
//: C03:Pitfall.cpp
// Operator mistakes

int main() {
    int a = 1, b = 1;
    while(a = b) {
        // ....
    }
}
```

```

    }
} ///:~

```

当**b**不为零时，语句**a = b**总是为真。把**b**的值赋给**a**，而**b**的值也是由运算符“=”产生的。一般在条件语句中，应当使用等值运算符“==”，而不是赋值。这是许多程序员经常犯的错误（但是，一些编译器会指出这个问题，这是有帮助的）。

一个相似的问题是使用位运算符中的“与”和“或”，而不是和它们相对应的逻辑运算符。位运算符中的“与”和“或”使用一个字符(& 或 |)，而逻辑“与”和“或”使用两个运算符(&&和||)。就像=和==一样，很容易会用一个字符替代两个字符。可以使用一种帮助记忆的方式“位比较小，所以在它们的运算符中不需要使用很多字符”。

### 3.7.11 转换运算符

转换 (cast) 这个词通常意为“浇铸成一个模型”。如果编译器能够明白的话，它会自动把一种数据类型转换为另一种类型。例如，如果赋一个整型值给一个浮点变量，编译器会暗地里调用一个函数（或更可能插入代码）来把整型转换为浮点型。转换允许使用这种显式类型变换，或在转换没有正常情况下发生时强制它实现。

为了实现转换，要用括号把所想要转换的数据类型（包括所有的修饰符）括起来放在值的左边。这个值可以是一个变量、一个常量、由一个表达式产生的值或是一个函数的返回值。下面是一个例子：

```

//: C03:SimpleCast.cpp
int main() {
    int b = 200;
    unsigned long a = (unsigned long int)b;
} ///:~

```

转换是很有用的，但是它也造成了令人头痛的事，因为在某些情况下，它强制编译器把一个数据看做是比它实际上更大的类型，所以它占用了更多的内存空间，这可能会破坏其他数据。这种情况经常不是出现在上述简单的类型转换时，而在转换指针时发生。

C++有一个另外的转换语法，它遵从函数调用的语法。这个语法给参数加上括号而不是给数据类型加上括号，类似于函数调用：

```

//: C03:FunctionCallCast.cpp
int main() {
    float a = float(200);
    // This is equivalent to:
    float b = (float)200;
} ///:~

```

当然对于上面的情况，我们实际上不需要转换，只要写**200f**（实际上，一般编译器会对上面的表达式作转换）。转换一般用于变量，而不用于常量。

### 3.7.12 C++的显式转换

应该小心使用转换，因为转换实际上要做的就是对编译器说“忘记类型检查，把它看做是其他类型。”这也就是说，在C++类型系统中引入了一个漏洞，并阻止编译器报告在类型方面出错了。更为糟糕的是，编译器会相信它，而不执行任何其他的检查来捕获错误。一旦

开始进行转换，程序员必须自己面对各种问题。事实上，无论什么原因，任何一个程序如果使用很多转换都值得怀疑。一般情况下，很少使用转换，它只是用于解决非常特殊的问题。

一旦理解了这一点，在遇上一个出故障的程序时，第一个反应应该是寻找作为嫌疑的转换。但是怎样确定C风格转换的位置呢？它们只是在括号中的类型名字，如果开始查找这些话，我们会发现很难把它们和代码的其他部分区分开来。

标准C++包括一个显式的转换语法，使用它来完全替代旧的C风格的转换（当然，如果不破坏代码，是不会认为C风格的转换不合法，但是编译器的编写者很容易标出旧风格的转换）。显式类型转换语法使我们很容易发现它们，因为通过它们的名字就能找到：

---

<b>static_cast</b>	用于“良性”和“适度良性”转换，包括不用强制转换（例如自动类型转换）
<b>const_cast</b>	对“const”和/或“volatile”进行转换
<b>reinterpret_cast</b>	转换为完全不同的意思。为了安全使用它，关键必须转换回原来的类型。转换成的类型一般只能用于位操作，否则就是为了其他隐秘的目的。这是所有转换中最危险的
<b>dynamic_cast</b>	用于类型安全的向下转换（这种转换将在第15章介绍）

---

在下面的小节会更详细地叙述前面三个显式转换，而最后一个要在读者有了更多的了解之后，在第15章中阐述。

#### 3.7.12.1 静态转换（static\_cast）

**static\_cast**全部用于明确定义的变换，包括编译器允许我们所做的不用强制转换的“安全”变换和不太安全但清楚定义的变换。**static\_cast**包含的转换类型包括典型的非强制变换、窄化（有信息丢失）变换，使用**void\***的强制变换、隐式类型变换和类层次的静态定位（因为还没有看到类和继承，这个主题会推延到第15章讨论）：

```
//: C03:static_cast.cpp
void func(int) {}

int main() {
    int i = 0x7fff; // Max pos value = 32767
    long l;
    float f;
    // (1) Typical castless conversions:
    l = i;
    f = i;
    // Also works:
    l = static_cast<long>(i);
    f = static_cast<float>(i);

    // (2) Narrowing conversions:
    i = l; // May lose digits
    i = f; // May lose info
    // Says "I know," eliminates warnings:
    i = static_cast<int>(l);
    i = static_cast<int>(f);
    char c = static_cast<char>(i);

    // (3) Forcing a conversion from void* :
    void* vp = &i;
    // Old way produces a dangerous conversion:
```





```

float* fp = (float*)vp;
// The new way is equally dangerous:
fp = static_cast<float*>(vp);
// (4) Implicit type conversions, normally
// performed by the compiler:
double d = 0.0;
int x = d; // Automatic type conversion
x = static_cast<int>(d); // More explicit
func(d); // Automatic type conversion
func(static_cast<int>(d)); // More explicit
} ///:~

```

程序的第(1)部分,是C中习惯采用的几种变换,有的有强制转换,有的没有强制转换。把int 提升到long或float不会有问题,因为后者总是能容纳一个int所包含的值。尽管这是不必要的,但是可以使用static\_cast来突出这些提升。

第(2)部分显示的是另一种变换方式。在这里可能会丢失数据,因为一个int和long或 float 不是一样“宽”的;它不能容纳同样大小的数字。因此称为窄化变换 (*narrowing conversion*)。编译器仍能执行这种转换,但是会经常给出一个警告。我们可以消除这种警告,表明我们真的想使用转换来实现它。

正如在第(3)部分看到的,C++中不用转换是不允许从void\*中赋值的(不像C)。这是很危险的,要求程序员知道他们正在做什么。至少,当查找故障的时候,static\_cast比旧标准的转换更容易定位。

程序的第(4)部分显示编译器自动执行的几种隐式类型变换。这些变换是自动的,不需要强制转换,但是当我们要想清楚发生了什么或以后要查找转换,可以再次使用static\_cast突出这个行为。

### 3.7.12.2 常量转换 (const\_cast)

如果从const转换为非const或从volatile转换为非volatile,可以使用const\_cast。这是const\_cast惟一允许的转换;如果进行别的转换就可能要使用单独的表达式或者可能会得到一个编译错误。

```

//: C03:const_cast.cpp
int main() {
    const int i = 0;
    int* j = (int*)&i; // Deprecated form
    j = const_cast<int*>(&i); // Preferred
    // Can't do simultaneous additional casting:
    //! long* l = const_cast<long*>(&i); // Error
    volatile int k = 0;
    int* u = const_cast<int*>(&k);
} ///:~

```

如果取得了const对象的地址,就可以生成一个指向const的指针,不用转换是不能将它赋给非const指针的。旧形式的转换能实现这样的赋值,但是const\_cast是适用的。volatile也是这样。

### 3.7.12.3 重解释转换 (reinterpret\_cast)

这是最不安全的一种转换机制,最有可能出问题。reinterpret\_cast把对象假想为模式(为了某种隐秘的目的),仿佛它是一个完全不同类型的对象。这是低级的位操作,C因此而名

声不佳。在使用`reinterpret_cast`做任何事之前，实际上总是需要`reinterpret_cast`回到原来的类型（或者把变量看做是它原来的类型）。

```
//: C03:reinterpret_cast.cpp
#include <iostream>
using namespace std;
const int sz = 100;

struct X { int a[sz]; };

void print(X* x) {
    for(int i = 0; i < sz; i++)
        cout << x->a[i] << ' ';
    cout << endl << "-----" << endl;
}

int main() {
    X x;
    print(&x);
    int* xp = reinterpret_cast<int*>(&x);
    for(int* i = xp; i < xp + sz; i++)
        *i = 0;
    // Can't use xp as an X* at this point
    // unless you cast it back:
    print(reinterpret_cast<X*>(xp));
    // In this example, you can also just use
    // the original identifier:
    print(&x);
} ///:~
```

在这个简单的例子中，**struct X**只包含一个整型数组，但是当用**X x**在堆栈中创建一个变量时，该结构体中的每一个整型变量的值都没有意义（通过使用函数`print()`把结构体的每一个整型值显示出来可以表明这一点）。为了初始化它们，取得**X**的地址并转换为一个整型指针，该指针然后遍历这个数组置每一个整型元素为0。注意**i**的上限是如何通过计算**sz**加**xp**得到的。编译器知道我们实际上是希望**sz**的指针位置比**xp**更大，它替我们做了正确的指针算术运算。

`reinterpret_cast`的思想就是当需要使用的時候，所得到的东西已经不同了，以至于它不能用于类型的原来目的，除非再次把它转换回来。这里，我们在打印调用中转换回**X\***，但是当然，因为我们还有原来的标识符，所以还可以使用它。但是**xp**只有作为**int\***才有用，这真的是对原来的**X**的重新解释。

使用`reinterpret_cast`通常是一种不明智、不方便的编程方式，但是当必须使用它时，它是非常有用的。

### 3.7.13 sizeof——独立运算符

**sizeof**单独作为一个运算符是因为它满足不同寻常的需要。**sizeof**给我们提供对有关数据项目所分配的内存大小。正如在本章前面叙述的那样，**sizeof**告诉我们任何变量使用的字节数。它也可以给出数据类型的大小（不用变量名）。

```
//: C03:sizeof.cpp
#include <iostream>
using namespace std;
```

```
int main() {
    cout << "sizeof(double) = " << sizeof(double);
    cout << ", sizeof(char) = " << sizeof(char);
} ///:~
```

按照定义,任何**char** (**signed**、**unsigned**或普通的)类型的**sizeof**都是1,不管**char**潜在的存储空间是否实际上是一个字节。对于所有别的类型,结果都是以字节表示的大小。

注意**sizeof**是一个运算符,不是函数。如果把它应用于一个类型,必须要像上面所示的那样使用括号,但是如果对一个变量使用它,可以不要括号。

```
//: C03:sizeofOperator.cpp
int main() {
    int x;
    int i = sizeof x;
} ///:~
```

**sizeof**也可以给出用户定义的数据类型的大小。这在本书后面会介绍。

### 3.7.14 asm关键字

这是一种转义 (escape) 机制,允许在C++程序中写汇编代码。在汇编程序代码中经常可以引用C++的变量,这意味着可以方便地和C++代码通信,且限定汇编代码只是用于必要的高效调整,或使用特殊的处理器指令。编写汇编语言时所必须使用的严格语法是依赖于编译器的,在编译器的文档中可以发现有关语法。

### 3.7.15 显式运算符

这是用于位运算符和逻辑运算符的关键字。没有**&**、**|**、**^**这些键盘字符的非美国程序员被迫使用C的令人讨厌的三个图形字符(*trigraph*),这使得不但在输入字符的时候令人烦恼,而且在阅读时也含义模糊。在C++中用附加的关键字来修补这种情况。

关 键 字	含 义
<b>and</b>	<b>&amp;&amp;</b> (逻辑与)
<b>or</b>	<b>  </b> (逻辑或)
<b>not</b>	<b>!</b> (逻辑非)
<b>not_eq</b>	<b>!=</b> (逻辑不等)
<b>bitand</b>	<b>&amp;</b> (位与)
<b>and_eq</b>	<b>&amp;=</b> (位与-赋值)
<b>bitor</b>	<b> </b> (位或)
<b>or_eq</b>	<b> =</b> (位或-赋值)
<b>xor</b>	<b>^</b> (位异或)
<b>xor_eq</b>	<b>^=</b> (位异或-赋值)
<b>compl</b>	<b>~</b> (补)

如果读者的编译器遵从标准C++, 它会支持这些关键字。

## 3.8 创建复合类型

基本的数据类型及其变体很重要,但也很简单。C和C++提供的工具允许把基本的数据类型组合成复杂的数据类型。正如我们将看到的那样,这些类型中最重要的是**struct**,在C++中

这是类的基础。但是，创建比较复杂的类型的最简单的一种方式，只需要通过**typedef**来命名一个名字为另一个名字。

### 3.8.1 用typedef命名别名

这个关键字从字面上看的作用比它实际所起的作用更大：**typedef**表示“类型定义”，但用“别名”来描述可能更精确，因为这正是它真正的作用。它的语法是：

**typedef** 原类型名 别名

当数据类型稍微有点复杂时，人们经常使用**typedef**只是为了少敲几个键。下面是一种经常使用的**typedef**：

```
typedef unsigned long ulong;
```

现在如果写**ulong**，则编译器知道意思是**unsigned long**。我们可能认为使用预处理程序置换就可以很容易实现，但是在一些重要的场合，编译器必须知道我们正在将名字当做类型处理，所以**typedef**起了关键作用。

**typedef**经常会派上用场的地方是指针类型。如前所述，如果写出

```
int*x, y;
```

这实际上生成一个**int\*x**和一个**int\*y**（不是一个**int\***）。也就是说，‘\*’绑定右边，而不是左边。但是，如果使用一个**typedef**：

```
typedef int* IntPtr;
IntPtr x, y;
```

则**x**和**y**都是**int\***类型。

有人可能争辩说避免使用**typedef**定义基本类型会更清楚，因此更可读，而使用大量**typedef**时，程序的确很快变得难以阅读。但是，在C中使用**struct**时，**typedef**是特别重要的。

### 3.8.2 用struct把变量结合在一起

**struct**（结构）是把一组变量组合成一个构造的一种方式。一旦创建了一个**struct**，就可以生成所建立的新类型变量的许多实例。例如：

```
//: C03:SimpleStruct.cpp
struct Structure1 {
    char c;
    int i;
    float f;
    double d;
};

int main() {
    struct Structure1 s1, s2;
    s1.c = 'a'; // Select an element using a '.'
    s1.i = 1;
    s1.f = 3.14;
    s1.d = 0.00093;
    s2.c = 'a';
    s2.i = 1;
    s2.f = 3.14;
```



```
s2.d = 0.00093;
} ///:~
```

**struct**的声明必须以分号结束。在`main()`中，创建了两个**Structure1**的实例：**s1**和**s2**。它们每一个都有各自独立的**c**、**i**、**f**和**d**版本。所以**s1**和**s2**表示了完全独立的变量块。要在**s1**或**s2**中选择一个元素，应该使用一个‘.’，使用C++ **class**对象的语法就是前面看到的那样，因为**class**对象是由**struct**演化而来的，所以**struct**是语法的来源。

注意这是使用**Structure1**的不便之处（正如所指出的那样，只是在C中需要，而不是C++）。在C中，当定义变量时，不能只说**Structure1**，必须说**struct Structure1**。这就是在C中使用**typedef**特别方便的地方。

```
//: C03:SimpleStruct2.cpp
// Using typedef with struct
typedef struct {
    char c;
    int i;
    float f;
    double d;
} Structure2;

int main() {
    Structure2 s1, s2;
    s1.c = 'a';
    s1.i = 1;
    s1.f = 3.14;
    s1.d = 0.00093;
    s2.c = 'a';
    s2.i = 1;
    s2.f = 3.14;
    s2.d = 0.00093;
} ///:~
```

当定义**s1**和**s2**时（但是注意它只有数据和特征，并不包括行为，这就是在C++中得到的真正的对象），通过这样使用**typedef**，可以假定**Structure2**是一个像**int**或**float**一样的内建类型（这是在C中；而在C++中，可以试图去掉**typedef**），我们将会看到，**struct**标识符已经脱离了原来的目的，因为这里的目的是创造**typedef**。当然，有时候可能需要早定义结构是使用**struct**。这时，可以重复**struct**的名字，就像**struct**名和**typedef**一样：

```
//: C03:SelfReferential.cpp
// Allowing a struct to refer to itself

typedef struct SelfReferential {
    int i;
    SelfReferential* sr; // Head spinning yet?
} SelfReferential;

int main() {
    SelfReferential sr1, sr2;
    sr1.sr = &sr2;
    sr2.sr = &sr1;
    sr1.i = 47;
    sr2.i = 1024;
} ///:~
```



如果看一下这个程序，会看到`sr1`和`sr2`互相指向且每个都拥有一块数据。

实际上，`struct`的名字不必和`typedef`的名字相同，但是，一般使用相同的名字，为了使得事物更加简单。

### 3.8.2.1 指针和`struct`

在上面的例子中，所有的`struct`都当做对象处理。但是，像任何一片存储空间一样，可以取得一个`struct`的地址（正如在上面的程序`SelfReferential.cpp`中看到的那样）。如上所述，为了选择一个特定`struct`对象中的元素，应当使用`'.'`。但是，如果有一个指向`struct`对象的指针，可以使用一个不同的运算符`'->'`来选择对象中的元素。下面是一个例子：

```
//: C03:SimpleStruct3.cpp
// Using pointers to structs
typedef struct Structure3 {
    char c;
    int i;
    float f;
    double d;
} Structure3;

int main() {
    Structure3 s1, s2;
    Structure3* sp = &s1;
    sp->c = 'a';
    sp->i = 1;
    sp->f = 3.14;
    sp->d = 0.00093;
    sp = &s2; // Point to a different struct object
    sp->c = 'a';
    sp->i = 1;
    sp->f = 3.14;
    sp->d = 0.00093;
} ///:~
```

在`main()`中，`struct`指针`sp`最初指向`s1`，用`'->'`选择`s1`中的成员来初始化它们。随后`sp`指向`s2`，以同样的方式初始化那些变量。所以可以看到指针的另一个好处是可以动态地重定向它们，指向不同的对象，使编程更灵活。

到现在为止，这就是对`struct`需要了解的全部，但是随着本书的进展，我们会更自如地使用它们（特别是它们更有潜力的继任者——类）。

### 3.8.3 用`enum`提高程序清晰度

枚举数据类型是把名字和数字相联系的一种方式，从而对阅读代码的任何人给出更多的含义。`enum`关键字（来自C）通过为所给出的任何标识符表赋值0、1、2等值来自动地列举出它们。也可以声明`enum`变量（它们总是表示为整数值）。`enum`的声明和`struct`的声明很相似。

当想明了某种特征时，枚举数据类型是很有用的：

```
//: C03:Enum.cpp
// Keeping track of shapes

enum ShapeType {
    circle,
    square,
```

```

    rectangle
}; // Must end with a semicolon like a struct

int main() {
    ShapeType shape = circle;
    // Activities here....
    // Now do something based on what the shape is:
    switch(shape) {
        case circle: /* circle stuff */ break;
        case square: /* square stuff */ break;
        case rectangle: /* rectangle stuff */ break;
    }
} //::~~

```

**shape**是被列举的数据类型**ShapeType**的变量，可以把它的值和列举的值相比较。因为**shape**实际上只是**int**，所以它可以具有任何一个**int**拥有的值（包括负数）。也可以把**int**变量和枚举值比较。

读者可能意识到上面的类型转换例子对于程序有可能是一种值得怀疑的方式。C++对这类程序有一种更好的编码方式，对它的解释在本书的后面介绍。

如果不喜欢编译器赋值的方式，可以自己做，如：

```

enum ShapeType {
    circle = 10, square = 20, rectangle = 50
};

```

如果对某些名字赋给值，对其他的不赋给值，编译器会使用相邻的下一个整数值。例如，

```
enum snap { crackle = 25, pop };
```

编译器会把值26赋给**pop**。

使用枚举数据类型时，增强了代码的可读性。然而，在某种程度上，这只是试图（在C中）实现在C++中用类可以做到的事，所以在C++中很少看到使用**enum**。

### 3.8.3.1 枚举类型检查

C的枚举相当简单，只是把整数值和名字联系起来，但它们并不提供类型检查。在C++中，正如现在希望的那样，类型的概念是基础，对于枚举也是如此。当创建一个命名的枚举时，就像使用类一样有效地创建了一个新类型。在单元翻译期间，枚举名成为保留字。

此外，在C++中对枚举的类型检查比在C中更为严格。如果有一个**color**枚举类型的实例**a**，我们会特别注意到这个。在C中，可以写**a++**，但在C++中不能这样写。这是因为枚举的增量运算执行两种类型转换，其中一个在C++中是合法的，另一个是不合法的。首先，枚举的值隐式地从**color**强制转换为**int**，然后递增该值，再把**int**强制转换回**color**类型。在C++中，这是不允许的，因为**color**是一个独特的类型，并不等价于一个**int**。这一点是有意义的，因为我们怎么能知道在颜色表中**blue**的增量值会是什么？如果想对**color**进行增量运算，则它应该是一个类（按照增量运算）而不是一个**enum**，成为一个类会更安全。任何时候写代码对**enum**类型进行隐式转换，编译器都会标记这是一个危险活动。

在C++中，联合（在下面描述）有很相似的附加类型检查。

### 3.8.4 用union节省内存

有时一个程序会使用同一个变量处理不同的数据类型。对于这种情况，有两种选择：可



以创建一个**struct**，其中包含需要存储的所有可能的不同类型，或者可以使用**union**（联合）。**union**把所有的数据放进一个单独的空间内，它计算出放在**union**中的最大项所必需的空间数，并生成**union**的大小。使用**union**可以节省内存。

每当在**union**中放置一个值，这个值总是放在**union**开始的同一个地方，但是只使用必需的空间。因此，我们创建的是一个能容纳任何一个**union**变量的“超变量”。所有的**union**变量地址都是一样的（在类或**struct**中，地址是不同的）。

下面是一个使用**union**的例子。试着去掉不同的元素，看看对**union**的大小有什么影响。注意在**union**中声明某个数据类型的多个实例是没有意义的（除非就是要用不同的名字）。

```
//: C03:Union.cpp
// The size and simple use of a union
#include <iostream>
using namespace std;

union Packed { // Declaration similar to a class
    char i;
    short j;
    int k;
    long l;
    float f;
    double d;
    // The union will be the size of a
    // double, since that's the largest element
}; // Semicolon ends a union, like a struct

int main() {
    cout << "sizeof(Packed) = "
          << sizeof(Packed) << endl;
    Packed x;
    x.i = 'c';
    cout << x.i << endl;
    x.d = 3.14159;
    cout << x.d << endl;
} ///:~
```

编译器根据所选择的联合的成员执行适当的赋值。

一旦进行赋值，编译器并不关心用联合做什么。在上面的例子中，可以对x赋一个浮点值：

```
x.f = 2.222;
```

然后把它作为一个**int**输出。

```
cout << x.i;
```

结果是无用的信息。

### 3.8.5 数组

数组是一种复合类型，因为它们允许在一个单一的标识符下把变量结合在一起，一个接一个。如果写出

```
int a[10];
```

就为10个**int**变量创建了一个接一个的存储空间，但是每一个变量并没有单独的标识符。相反，

它们都集结在名字**a**下。

要访问一个数组元素，可以使用定义数组时所使用的方括号语法：

```
a[5] = 47;
```

不过，必须记住，尽管**a**的大小是10，但是要从零开始选择数组元素（有时这被称为零指针），所以只可以选择数组元素0~9，如下所示：

```
//: C03:Arrays.cpp
#include <iostream>
using namespace std;

int main() {
    int a[10];
    for(int i = 0; i < 10; i++) {
        a[i] = i * 10;
        cout << "a[" << i << "] = " << a[i] << endl;
    }
} ///:~
```

访问数组是很快的。但是，如果下标超出数组的界限，这就不安全了，这可能会访问到别的变量。另一个缺陷是必须在编译期定义数组的大小；如果想在运行期改变大小，则不能使用上面的语法（C有一种动态创建数组的方式，但是这会造成严重的混乱）。在前面一章中介绍的C++向量提供了类似数组的对象，它能自动调整自身的大小，所以如果数组的大小在编译期不能确定的话，这是比较好的解决方法。

可以生成任何类型的数组，甚至是**struct**类型的：

```
//: C03:StructArray.cpp
// An array of struct

typedef struct {
    int i, j, k;
} ThreeDpoint;

int main() {
    ThreeDpoint p[10];
    for(int i = 0; i < 10; i++) {
        p[i].i = i + 1;
        p[i].j = i + 2;
        p[i].k = i + 3;
    }
} ///:~
```

注意：**struct**中的标识符**i**如何与**for**循环中的**i**无关。

为了知道数组中的相邻元素之间的距离，可以打印出地址如下：

```
//: C03:ArrayAddresses.cpp
#include <iostream>
using namespace std;

int main() {
    int a[10];
    cout << "sizeof(int) = " << sizeof(int) << endl;
    for(int i = 0; i < 10; i++)
        cout << "&a[" << i << "] = "
```



```

        << (long)&a[i] << endl;
    } ///:~

```

当运行程序时，会看到每一个元素和前一个元素都是相距`int`大小的距离。也就是说，它们是一个接一个存放的。

### 3.8.5.1 指针和数组

数组的标识符不像一般变量的标识符。一方面，数组标识符不是左值，不能给它赋值。它只是一个进入方括号语法的手段，当给出数组名而没有方括号时，得到的就是数组的起始地址：

```

//: C03:ArrayIdentifier.cpp
#include <iostream>
using namespace std;

int main() {
    int a[10];
    cout << "a = " << a << endl;
    cout << "&a[0] =" << &a[0] << endl;
} ///:~

```

运行这个程序时，会看到这两个地址（因为没有转换为`long`，所以它以十六进制的形式打印出来）是一样的。

因此可以把数组标识符看做是数组起始处的只读指针。尽管不能改变数组标识符指向，但是可以另创建指针，使它在数组中移动。事实上，方括号语法和指针一样工作：

```

//: C03:PointersAndBrackets.cpp
int main() {
    int a[10];
    int* ip = a;
    for(int i = 0; i < 10; i++)
        ip[i] = i * 10;
} ///:~

```

当想给一个函数传递数组时，命名数组以产生它的起始地址的事实相当重要。如果声明一个数组为函数参数，实际上真正声明的是一个指针。所以在下面的例子中，`func1()`和`func2()`有一样的参数表：

```

//: C03:ArrayArguments.cpp
#include <iostream>
#include <string>
using namespace std;

void func1(int a[], int size) {
    for(int i = 0; i < size; i++)
        a[i] = i * i - i;
}

void func2(int* a, int size) {
    for(int i = 0; i < size; i++)
        a[i] = i * i + i;
}

void print(int a[], string name, int size) {
    for(int i = 0; i < size; i++)

```



```

        cout << name << "[" << i << "]" = "
            << a[i] << endl;
    }

    int main() {
        int a[5], b[5];
        // Probably garbage values:
        print(a, "a", 5);
        print(b, "b", 5);
        // Initialize the arrays:
        func1(a, 5);
        func1(b, 5);
        print(a, "a", 5);
        print(b, "b", 5);
        // Notice the arrays are always modified:
        func2(a, 5);
        func2(b, 5);
        print(a, "a", 5);
        print(b, "b", 5);
    } ///:~

```

尽管**func1()**和**func2()**以不同的方式声明它们的参数<sup>⑨</sup>，但是在函数内部的用法是一样的。这个例子暴露出了一些别的问题：数组不可以按值传递，也就是说，不会自动地得到传递给函数的数组的本地拷贝。因此，修改数组时，一直是在修改外部对象。如果想按照一般的参数那样提供按值传递，可能一开始会让人有点迷惑。

读者会注意到，**print()**对数组参数使用方括号语法。尽管把数组作为参数传递时，指针语法和方括号语法是一样的，但是方括号语法使得读者更清楚它的意思是把这个参数看做是一个数组。

还要注意，在每一种情况传递了参数**size**。仅仅传递数组的地址还不能提供足够的信息，必须知道在函数中的数组有多大，这样就不会超出数组的界。

数组可以是任何一种类型，包括指针数组。事实上，想给程序传递命令行参数时，C和C++的函数**main()**有特殊的参数表，其形式如：

```
int main(int argc, char* argv[]) { // ...
```

第一个参数的值是第二个参数的数组元素个数。第二个参数总是**char\***数组，因为数组中的元素来自作为字符数组的命令行（记住，数组只能作为指针传递）。命令行中的每一个用空格分隔的字符串被转换成单独的数组参数。通过遍历数组，下面的程序可以打印出所有的命令行参数：

```

//: C03:CommandLineArgs.cpp
#include <iostream>
using namespace std;

int main(int argc, char* argv[]) {
    cout << "argc = " << argc << endl;

```

⑨ 除非采取了严格的办法：“在C/C++中的所有参数是通过值传递的，数组的‘值’是由数组标识符产生的；它是一个地址。”从汇编语言的观点来看这可能是真的，但是当用更高层的概念工作时，我认为这是没有帮助的。在C++中附加的引用生成“所有的传递都是通过值”的说法更会使人混淆，对于这一点我认为按照与“以地址传递”相对的“以值传递”来思考更好。

```

    for(int i = 0; i < argc; i++)
        cout << "argv[" << i << "] = "
            << argv[i] << endl;
} ///:~

```

读者会注意到`argv[0]`是程序本身的路径和名字。它允许程序发现自己的信息。它也给程序参数数组增加一个或多个参数，所以一个常见的错误就是当想获取命令行参数`argv[1]`的值时，却去取`argv[0]`的值。

在函数`main()`中，不要强制使用`argc`和`argv`为标识符；这些标识符只是习惯用法（如果不使用它们，可能会让别人迷惑）。还有另一种声明`argv`的方式：

```
int main(int argc, char** argv) { // ...
```

两种形式是等价的，但本书使用的版本更为直观，因为它直接表明“这是一个字符指针数组”。

从命令行中获得的是字符数组；如果想把数组看成是别的某种类型，应该在程序里负责转换它。为了便于转换为数值，在标准C库的`<cstdlib>`中声明了一些更有帮助的函数。最简单的是分别使用`atoi()`、`atol()`和`atof()`把ASCII字符数组转换为`int`、`long`和`double`浮点值。下面是一个使用`atoi()`的例子（另两个函数用同样的方式调用）：

```

//: C03:ArgsToInts.cpp
// Converting command-line arguments to ints
#include <iostream>
#include <cstdlib>
using namespace std;

int main(int argc, char* argv[]) {
    for(int i = 1; i < argc; i++)
        cout << atoi(argv[i]) << endl;
} ///:~

```

在这个程序中，可以在命令行中放置任意多个参数。读者会注意到`for`循环从值1开始，跳过了`argv[0]`中的程序名。如果在命令行上放置了一个包含小数点的浮点数，`atoi()`只取得小数点前面的数字部分。如果在命令行中没有数值，`atoi()`会返回零值。

### 3.8.5.2 探究浮点格式

本章已经介绍的`printBinary()`函数对于研究不同数据类型的内部结构是很合适的。最令人感兴趣的还是浮点格式，它允许C和C++在有限的空间里存储非常大和非常小的数。尽管在这里不能完全显示其细节，但是在`float`和`double`里的数字位被分为段：指数、尾数和符号位，它用科学计数法来存储数值。下面的程序允许打印出不同浮点数的二进制形式，所以读者可以自己推断出编译器浮点格式的使用方案（一般这是浮点数的IEEE标准，但是有的编译器可能不遵守）。

```

//: C03:FloatingAsBinary.cpp
//{L} printBinary
//{T} 3.14159
#include "printBinary.h"
#include <cstdlib>
#include <iostream>
using namespace std;

```

```

int main(int argc, char* argv[]) {
    if(argc != 2) {
        cout << "Must provide a number" << endl;
        exit(1);
    }
    double d = atof(argv[1]);
    unsigned char* cp =
        reinterpret_cast<unsigned char*>(&d);
    for(int i = sizeof(double); i > 0 ; i -= 2) {
        printBinary(cp[i-1]);
        printBinary(cp[i]);
    }
} ///:~

```

首先，程序通过检查**argc**的值保证给定了参数，如果有一个参数，则**argc**的值应该为2（如果没有参数，则为1，因为程序名总是**argv**的第一个元素）。如果程序失败了，会打印出一个消息并调用标准C的库函数**exit()**来终止程序。

程序从命令行中取得参数并使用函数**atof()**把字符转换成**double**浮点数。然后通过取得地址并把该数转换为一个**unsigned char\***指针作为一个字节数组。把其中的每一个字节传递给**printBinary()**显示出来。

我在自己的机器上通过了这个程序，打印字节时符号位出现在前面。有的机器可能和我的不一样，所以可能需要重新安排打印的方式。读者应认识到理解浮点格式并不是微不足道的。例如，一般不把指数和尾数以字节划分的边界存放，而是为每一部分保留若干位数，并把它们尽可能紧密地压缩进内存。要真的看看发生了什么，应该把数值的每一部分的大小找出来（符号位总是一位，而指数和尾数的位数的大小不同），并把每一部分的位数分别打印出来。

### 3.8.5.3 指针算术

如果用指针所做的工作只是把它看做是数组的一个别名，那么指向数组的指针可能不太令人感兴趣。但是，指针比这个更灵活，因为可以修改它们指向任何别的地方（但是记住，不能修改数组标识符来指向别的地方）。

指针算术（*pointer arithmetic*）指的是对指针的某些算术运算符的应用。指针算术是一个源自普通算术的单独主题，其原因在于为了正确运行，指针必须遵守特定的约束。例如，指针常用的运算符是**++**——“给指针加1”。它的实际意义是改变指针移向“下一个值”。下面是一个例子：

```

//: C03:PointerIncrement.cpp
#include <iostream>
using namespace std;

int main() {
    int i[10];
    double d[10];
    int* ip = i;
    double* dp = d;
    cout << "ip = " << (long)ip << endl;
    ip++;
    cout << "ip = " << (long)ip << endl;
    cout << "dp = " << (long)dp << endl;
    dp++;
    cout << "dp = " << (long)dp << endl;
} ///:~

```



我的机器上的运行输出是：

```
ip = 6684124
ip = 6684128
dp = 6684044
dp = 6684052
```

这里令人感兴趣的是尽管对**int\***和**double\***进行的都是同样的操作“++”，但是对**int\***只改变了4个字节，而对**double\***改变了8个字节。当然并非总是这样，这取决于**int**和**double**浮点数的大小。这就是指针算术的技巧：编译器计算出指针改变的正确值，使它指向数组中的下一个元素（指针算术只有在数组中才是有意义的）。甚至在**struct**数组中也能这样工作：

```
//: C03:PointerIncrement2.cpp
#include <iostream>
using namespace std;

typedef struct {
    char c;
    short s;
    int i;
    long l;
    float f;
    double d;
    long double ld;
} Primitives;

int main() {
    Primitives p[10];
    Primitives* pp = p;
    cout << "sizeof(Primitives) = "
          << sizeof(Primitives) << endl;
    cout << "pp = " << (long)pp << endl;
    pp++;
    cout << "pp = " << (long)pp << endl;
} ///:~
```

我的机器上的运行结果是：

```
sizeof(Primitives) = 40
pp = 6683764
pp = 6683804
```

所以可以看到编译器对于**struct**（以及**class**和**union**）指针也能正确地工作。

指针算术运算也可以使用运算符“--”、“+”和“-”，但是后面两个运算符的使用是有限制的：不能把两个指针相加，如果使指针相减，其结果是两个指针之间相隔的元素个数。不过，一个指针可以加上或减去一个整数。下面是一个说明指针算术运算用法的例子：

```
//: C03:PointerArithmetic.cpp
#include <iostream>
using namespace std;

#define P(EX) cout << #EX << ": " << EX << endl;

int main() {
    int a[10];
```



```

    for(int i = 0; i < 10; i++)
        a[i] = i; // Give it index values
    int* ip = a;
    P(*ip);
    P(++ip);
    P(*(ip + 5));
    int* ip2 = ip + 5;
    P(*ip2);
    P(*(ip2 - 4));
    P(--ip2);
    P(ip2 - ip); // Yields number of elements
} ///:~

```

这个程序以另一个宏开始，但是它使用了一个被称为字符串化的预处理器特征（在表达式前用一个‘#’实现），其作用是获得任何一个表达式并把它转换成为一个字符数组。这是很方便的，因为它允许打印一个表达式，后面接一个冒号，再接一个表达式的值。在`main()`中，可以看到这产生了一个有用的简化。

尽管`++`和`--`的前缀和后缀方式对指针来说都是有效的，但是在这个例子中只使用了前缀方式，因为在上面的表达式中指针间接引用之前先应用它们，所以它们允许看到运算的效果。注意只能加上和减去整数值，如果两个指针以这种方式结合，编译器是不允许的。

上面程序的输出是：

```

*ip: 0
*++ip: 1
*(ip + 5): 6
*ip2: 6
*(ip2 - 4): 2
*--ip2: 5

```

在各种情况下，指针算术根据所指元素的大小调整指针，使其指向“正确的地方”。

如果一开始指针算术运算看起来有点令人困扰，那么不必担心。大多数情况下只需要创建数组和用`[]`表示的数组下标，一般所需要的最为复杂的指针算术运算是`++`和`--`。指针运算一般都用于更为灵活和复杂的程序中，标准C++库中许多容器隐藏了大多数的灵活细节，所以不必担心这一点。

## 3.9 调试技巧

在理想环境下，因为有优秀的调试器能很容易使得程序的运行行为透明，所以可以很快发现错误。但是，大多数的调试器都有盲点，这就需要在程序中插入小段代码来帮助理解发生了什么。此外，可能在没有调试器（例如一个嵌入式系统）或者可能只有少量的反馈（如一个单行的LED显示屏）的环境下进行开发。在这些情况下，就要用创造性的方法去发现和显示关于程序执行情况的信息。下一节对程序调试的技巧提出某些建议。

### 3.9.1 调试标记

如果在程序中加入调试代码，可能引起不便。一开始得到了太多的信息，这使得很难把故障孤立出来。当认为已经找到了故障时，我们开始删掉调试代码，却有可能发现再需要这些代码。我们可以用两种标记解决这类问题：预处理器调试标记和运行期调试标记。

### 3.9.1.1 预处理器调试标记

通过使用预处理器**#define**定义一个或更多的调试标记（在头文件中更适合），可以测试一个使用**#ifdef**语句和包含条件调试代码的标记。当认为调试完成了，只需使用**#undef**标记，代码就会自动消失（这会减少可执行文件的大小和运行时间）。

最好在开始建立工程前决定调试标记的名字，这样名字会一致。为了区分预处理器标记和变量，预处理器标记一般用大写字母书写。一个常用的标记名是**DEBUG**（但是小心，不能使用**NDEBUG**，它是C中的保留字）。语句序列可以是：

```
#define DEBUG // Probably in a header file
//...
#ifdef DEBUG // Check to see if flag is defined
/* debugging code here */
#endif // DEBUG
```

大多数C和C++的程序实现还允许在编译器的命令行中使用**#define**和**#undef**标记，所以可以用一个单独的命令重新编译代码并插入调试信息（最好使用makefile，这是后面要简要说明的工具）。具体细节请看局部的文档。

### 3.9.1.2 运行期调试标记

在某些情况下，在程序执行期间打开和关闭调试标记会更加方便，特别是使用命令行在启动程序时设置它们。只是为了插入调试代码来重新编译一个大程序是很乏味的。

为了自动打开和关闭调试代码，可以建立一个如下的**bool**标记：

```
//: C03:DynamicDebugFlags.cpp
#include <iostream>
#include <string>
using namespace std;
// Debug flags aren't necessarily global:
bool debug = false;

int main(int argc, char* argv[]) {
    for(int i = 0; i < argc; i++)
        if(string(argv[i]) == "--debug=on")
            debug = true;
    bool go = true;
    while(go) {
        if(debug) {
            // Debugging code here
            cout << "Debugger is now on!" << endl;
        } else {
            cout << "Debugger is now off." << endl;
        }
        cout << "Turn debugger [on/off/quit]: ";
        string reply;
        cin >> reply;
        if(reply == "on") debug = true; // Turn it on
        if(reply == "off") debug = false; // Off
        if(reply == "quit") break; // Out of 'while'
    }
}
//::~~
```

这个程序一直允许打开和关闭调试标记，直到输入“quit”告诉它想要退出。注意需要输入整个单词，而不仅仅是字母（如果想要的话，可以缩写它为字母）。在启动时，可以选择性

地使用命令行参数打开调试——这个参数可以出现在命令行的任意地方，因为`main()`中的启动代码能看得到所有的参数。测试是相当简单的，因为表达式为：

```
string(argv[i])
```

取得`argv[i]`字符数组并创建一个`string`使得它容易和`==`右端比较。上面的程序查找整个字符串`--debug=on`。也可以寻找`--debug=`，然后看它后面有什么，以提供更多的选择。本书的第2卷（可从[www.BruceEckel.com](http://www.BruceEckel.com)中获得）有专门的一章讲述标准C++ `string`类。

虽然调试标记是很少的几个领域之一，其中对于使用全局变量很有意义，但是，并不是说必须这样做。注意使用小写字母书写变量，用来提醒读者它不是一个预处理器标记。

### 3.9.2 把变量和表达式转换成字符串

写调试代码的时候，编写由包含变量名和后跟变量的字符数组组成的打印表达式是很乏味的。幸运的是，标准C具有字符串化运算符`#`，它在本章前面使用过的。在一个预处理器宏中的参数前面使用一个`#`，预处理器会把这个参数转换为一个字符数组。把这一点与没有插入标点符号的若干个字符数组结合而连接成一个单独的字符数组，能够生成一个十分方便的宏用于调试期间打印出变量的值：

```
#define PR(x) cout << #x " = " << x << "\n";
```

如果调用宏`PR(a)`来打印变量`a`的值，它和下面的代码有同样的效果：

```
cout << "a = " << a << "\n";
```

整个表达式工作过程一样。下面的程序使用一个宏创建了一种速记方式打印出字符串化的表达式，然后计算表达式并打印出结果：

```
//: C03:StringizingExpressions.cpp
#include <iostream>
using namespace std;

#define P(A) cout << #A << ": " << (A) << endl;

int main() {
    int a = 1, b = 2, c = 3;
    P(a); P(b); P(c);
    P(a + b);
    P((c - a)/b);
} ///:~
```

可以看到像这样的技术是如何成为必不可少的，特别是在没有调试器（或者必须使用多个开发环境）的情况下。当不想调试时，也可以插入一个`#ifdef`使得定义的`P(A)`不起作用。

### 3.9.3 C语言`assert()`宏

在标准头文件`<cassert>`中，会发现`assert()`是一个方便的调试宏。当使用`assert()`时，给它一个参数，即一个表示断言为真的表达式。预处理器产生测试该断言的代码。如果断言不为真，则在发出一个错误信息告诉断言是什么以及它失败之后，程序会终止。下面是一个例子：

```
//: C03:Assert.cpp
// Use of the assert() debugging macro
```

```
#include <cassert> // Contains the macro
using namespace std;

int main() {
    int i = 100;
    assert(i != 100); // Fails
} ///:~
```

这个宏来源于标准C，所以在头文件**assert.h**中也可以使用。

当完成调试后，通过在程序的**#include<cassert>**之前插入语句行

```
#define NDEBUG
```

或者在编译器命令行中定义**ndebug**，可以消除宏产生的代码。在**<cassert>**中使用的**ndebug**是一个标记，用来改变宏产生代码的方式。

在本书后面，会看到对于**assert()**有一些更复杂的可供选择的方式。

### 3.10 函数地址

一旦函数被编译并载入计算机中执行，它就会占用一块内存。这块内存有一个地址，因此函数也有地址。

可以通过指针使用函数地址，就像可以使用变量的地址一样。函数指针的声明和使用初看起来有点模糊，但是它同语言其余部分的格式一致。

#### 3.10.1 定义函数指针

要定义一个指针指向一个无参无返回值的函数，可以写成：

```
void (*funcPtr)();
```

当看到像这样的一个复杂定义时，最好的处理方法是中间开始和向外扩展。“从中间开始”的意思是从变量名开始，这里是指**funcPtr**。“向外扩展”的意思是先注意右边最近的项（在这个例子中没有该项，以右括号结束），然后注意左边（用星号表示的指针），注意右边（空参数表表示这个函数没有带任何参数），再注意左边（**void**指示函数没有返回值）。大多数声明都是以右-左-右动作的方式工作的。

回过头来看，“中间开始”（“**funcPtr**是一个...”），向右边走（没有东西，被右括号拦住了），向左边走并发现一个“\*”（“...指针指向一个...”），向右边走并发现一个空参数表（“...没有带参数的函数...”），向左边走并发现一个**void**（“**funcPtr**是一个指针，它指向一个不带参数并返回**void**的函数”）。

读者可能感到奇怪为什么**\*funcPtr**需要括号。如果不使用括号，编译器会看到：

```
void *funcPtr();
```

这可能是在声明一个函数（返回一个**void\***）而不是定义一个变量。在了解一个声明和定义应该是什么的时候，可以想象编译器要经历同样的过程。所以要“遇到”这些括号，使得编译器会返回左边并发现“\*”，而不是一直向右发现一个空参数表。

#### 3.10.2 复杂的声明和定义

另一方面，一旦知道C和C++声明语法是如何工作的，就能够创建许多复杂的条目。例如：

```

//: C03:ComplicatedDefinitions.cpp

/* 1. */      void * (*fp1)(int))[10];

/* 2. */      float (*fp2)(int,int,float))(int);

/* 3. */      typedef double (*(*fp3)())[10])();
               fp3 a;

/* 4. */      int (*f4())[10])();

int main() {} ///:~

```

对于每一条，使用先右后左的原则去推断。

第1行说明：“**fp1**是一个指向函数的指针，该函数接受一个整型参数并返回一个指向含有10个**void**指针数组的指针。”

第2行说明：“**fp2**是一个指向函数的指针，该函数接受三个参数（**int**、**int**和**float**）且返回一个指向函数的指针，该函数接受一个整型参数并返回一个**float**。”

如果创建许多复杂的定义，可以使用**typedef**。第3行显示了每次**typedef**是如何缩短复杂定义的。它说明：“**fp3**是一个指向函数的指针，该函数无参数，且返回一个指向含有10个指向函数指针数组的指针，这些函数不接受参数且返回**double**值。”然后它又说明：“**a**是**fp3**类型中的一个。”**typedef**在用简单描述构建复杂描述时通常是很有用的。

第4行不是变量定义而是一个函数定义。它说明：“**f4**是一个返回指针的函数，该指针指向含有10个函数指针的数组，这些函数返回整型值。”

我们可能很少甚至是从未使用过如此复杂的声明和定义。但如果通过练习能把它搞清楚的话，就不会被在现实生活中可能遇到的稍微复杂的情况所困惑。

### 3.10.3 使用函数指针

一旦定义了一个函数指针，在使用前必须给它赋一个函数的地址。就像一个数组**arr[10]**的地址是由不带方括号的这个数组的名字(**arr**)产生的一样，函数**func()**的地址也是由没有参数列表的函数名(**func**)产生的。也可以使用更加明显的语法**&func()**。为了调用这个函数，应当用与声明相同的方法间接引用指针。（记住，C和C++总是力图让引用看上去与使用它们的方法一样。）下面的例子表明如何定义和使用指向函数的指针：

```

//: C03:PointerToFunction.cpp
// Defining and using a pointer to a function
#include <iostream>
using namespace std;

void func() {
    cout << "func() called..." << endl;
}

int main() {
    void (*fp)(); // Define a function pointer
    fp = func;    // Initialize it
    (*fp)();      // Dereferencing calls the function
    void (*fp2)() = func; // Define and initialize
}

```



```

    (*fp2)();
} ///:~

```

在定义了指向函数的指针`fp`之后，用`fp = func`使`fp`获得函数`func()`的地址（注意在函数名后缺少了参数列表）。第二种情况显示了同时定义和初始化。

### 3.10.4 指向函数的指针数组

我们能够创建的一个更为有趣的结构是指向函数的指针数组。为了选择一个函数，只需要使用数组的下标，然后间接引用这个指针。这种方式支持表格式驱动码（*table-driven code*）的概念；可以根据状态变量（或者状态变量的组合值）去选择被执行函数，而不用条件语句或`case`语句。这种设计方式对于经常要从表中添加或删除函数（或者想动态地创建或改变表）十分有用。

下面的例子使用预处理宏创建了一些哑函数，然后使用自动聚合初始化功能创建指向这些函数的指针数组。正如看到的那样，很容易从表中添加或删除函数（这样，这个程序就具有了函数功能）而只需改变少量的代码：

```

//: C03:FunctionTable.cpp
// Using an array of pointers to functions
#include <iostream>
using namespace std;

// A macro to define dummy functions:
#define DF(N) void N() { \
    cout << "function " #N " called..." << endl; }

DF(a); DF(b); DF(c); DF(d); DF(e); DF(f); DF(g);

void (*func_table[])() = { a, b, c, d, e, f, g };

int main() {
    while(1) {
        cout << "press a key from 'a' to 'g' "
              "or q to quit" << endl;
        char c, cr;
        cin.get(c); cin.get(cr); // second one for CR
        if ( c == 'q' )
            break; // ... out of while(1)
        if ( c < 'a' || c > 'g' )
            continue;
        (*func_table[c - 'a'])();
    }
} ///:~

```

当希望创建一些解释器或表处理程序时，可以想象这种技术是多么有用。

## 3.11 make: 管理分段编译

当使用分段编译（*separate compilation*）（把代码拆分为许多翻译单元）时，需要某种方法去自动编译每个文件并且告诉连接器把所有分散的代码段，连同适当的库和启动代码，构造成一个可执行的文件。许多编译器允许用一个简单的命令行语句完成。例如，对于GNU C++编译器，可能会用：

```
g++ SourceFile1.cpp SourceFile2.cpp
```

使用这种方法的问题是编译器事先要编译每个文件而不管文件是否需要重建。在具有多个文件的工程中，如果仅仅改变了一个文件，就可能不得不重新编译所有文件。

解决问题的方法是用一个称为**make**的程序。该程序是在UNIX上开发的，但某些形式到处都可使用。**make**工具按照一个名为**makefile**的文本文件中的指令去管理一个工程中的所有单个文件。当编辑了工程中的某些文件并使用**make**时，**make**程序会按照**makefile**中的说明去比较源代码文件与相应目标文件的日期，如果源代码文件的日期比它的目标文件的日期新，**make**会调用编译器对源代码进行编译。**make**仅仅编译已经改变了的源代码，以及其他受修改文件影响的源代码文件。使用**make**程序，每次修改程序时，不必重新编译工程中的所有文件，也不必核对所有生成的东西。**makefile**文件包含了组合工程的所有命令。学会使用**make**命令会节省大量时间，也会减少挫折。在Linux/Unix机器上安装新软件时使用**make**是一种典型的方式(虽然那些**makefile**比本书上出现的要复杂得多，而且作为安装过程的一部分，对于特定的机器，通常会自动地生成**makefile**文件)。

因为**make**实际上对所有C++编译器有某种可用的形式(即使没有，也可以在任何编译器上使用免费的**make**)，因此它将作为贯穿于本书的工具。然而，编译器提供商也创建了自己的工程构造工具。这些工具询问工程中包括哪些文件，然后它们确定所有的关系。这些工具使用与**makefile**相似的文件，通常称为工程文件(*project file*)，程序环境会维护该文件因此不必为它而担心。配置和使用工程文件随开发环境的改变而有所不同，因此必须找到怎样使用它们的相关文档(虽然工程文件工具由不同的厂商提供，但是使用都很简单)。

即使还使用特定厂商的构建工程工具，本书中所用的**makefile**仍然有效。

### 3.11.1 make的行为

当输入**make**(或你的“**make**”程序的其他名字)时，**make**程序在当前目录中寻找名为**makefile**的文件，该文件作为工程文件已经被建立。这个文件列出了源代码文件间的依赖关系。**make**程序观察文件的日期。如果一个依赖文件的日期比它所依赖的文件旧，**make**程序执行依赖关系之后列出的规则。

在**makefile**中的所有注释都从“#”开始一直延续到本行的末尾。

作为一个简单的例子，一个名为“hello”的程序的**makefile**文件可能包含：

```
# A comment
hello.exe: hello.cpp
    mycompiler hello.cpp
```

这就是说**hello.exe**(目标文件)依赖于**hello.cpp**。当**hello.cpp**比**hello.exe**文件日期新时，**make**执行“规则”**mycompiler hello.cpp**。可能会有多重依赖和多重规则。许多**make**程序要求所有规则以tab开头。这与空格通常被忽略的空格不一样，空格可以用于格式化以便于阅读。

规则不仅局限于调用编译器。在**make**中还可以调用想要调用的任何程序。通过创建相互依赖的规则集的分组，可以修改源代码文件，输入**make**，确信所有受影响的文件会重新正确地重建。

#### 3.11.1.1 宏

**makefile**可以包含某些宏(注意，这些宏完全不同于C/C++的预处理宏)。用宏进行字符



串替换是很方便的。本书中的**makefile**使用一个宏去调用C++编译器。例如，

```
CPP = mycompiler
hello.exe: hello.cpp
    $(CPP) hello.cpp
```

等号‘=’用来把**CPP**定义为一个宏，符号‘\$’和圆括号扩展宏。在这里，扩展意味着宏调用**\$(CPP)**将被字符串**mycompiler**取代。对于上面的宏，如果想改变到名为**cpp**的不同编译器，只需把宏改变为：

```
CPP = cpp
```

也可以在宏中加入编译器标志，或使用分开的宏加入编译器标志。

### 3.11.1.2 后缀规则

说明**make**怎样为工程中的每个单独的**cpp**文件调用编译器是很乏味的，特别当知道了每次相同的处理过程之后。因为**make**的设计注重节约时间，所以只要依赖于文件名字后缀，它就有一种简化操作的方式。这些简化称为后缀规则。一条后缀规则是一种教**make**怎样从一种类型文件（如**cpp**）转化为另一种类型（如**obj**或**exe**）的方法。一旦有了**make**从一种文件转化为另外一种文件的规则，其他要做的只是告诉**make**哪些文件依赖于其他文件。当**make**发现一个文件比它依赖的文件旧，它就会使用规则创建一个新文件。

后缀规则告诉**make**可以根据文件的扩展名去考虑怎样构建程序而不需用显式规则去构建一切。在这种情况下它指出：“调用下面的命令从扩展名为**cpp**的文件去构造扩展名为**exe**的文件”。上述例子看起来如以下所示：

```
CPP = mycompiler
.SUFFIXES: .exe .cpp
.cpp.exe:
    $(CPP) $<
```

**.SUFFIXES**指令告诉**make**必须注意后面的扩展名，因为它们对于这个特定的**makefile**有特殊的意义。其后看到后缀规则**cpp.exe**，说明“这里是怎样把任何扩展名为**cpp**的文件转化为一个扩展名为**exe**的文件的”（当**cpp**文件比**exe**文件新的时候）。和前面一样使用了宏**\$(CPP)**，但是发现了某种新东西：**\$<**。因为以‘\$’开头，所以这是一个宏，但它是**make**内部的特殊的宏。符号**\$<**只能用于后缀规则，意思是“无论怎样都要触发的规则”（有时称为依赖），在本例中表示“需要被编译的**cpp**文件。”

一旦建立了后缀规则，就能简单地说明，例如说明“**make Union.exe**”，后缀规则会展开，即使在整个**makefile**文件中从未提及“**Union**”。

### 3.11.1.3 默认目标

在宏和后缀规则之后，**make**在文件中查找第一个“目标”，并构建它，除非指定了不同的目标文件。因此对于**makefile**文件：

```
CPP = mycompiler
.SUFFIXES: .exe .cpp
.cpp.exe:
    $(CPP) $<
target1.exe:
target2.exe:
```

如果简单地输入‘**make**’，那么会生成**target1.exe**文件（使用默认的后缀规则），因为它是

**make**遇到的第一个目标。为了生成**target2.exe**我们不得不显式说明‘**make target2.exe**’。这样做就比较冗长，因此通常会创建一个依赖于所有其余目标文件的默认“哑元”目标，例如：

```

CPP = mycompiler
.SUFFIXES: .exe .cpp
.cpp.exe:
    $(CPP) $<
all: target1.exe target2.exe

```

在这里，‘**all**’并不存在，没有名为‘**all**’的文件，因此每次键入**make**，它会把‘**all**’作为第一个目标（这是默认的目标），然后发现‘**all**’不存在，所以它检查所有的依赖关系。因此它查看**target1.exe**并（使用后缀规则）判断：(1) **target1.exe**文件是否存在，(2) **target1.cpp**文件是否比**target1.exe**文件新。如果(1) (2)都成立，就使用后缀规则（除非为某个特定的文件提供了一个显式规则）。然后在默认的目标列表上查找下一个目标文件。因此通过建立一个默认的目标文件列表（按习惯通常称为‘**all**’，但可以随便起名），只需简单地键入**make**就能够生成在工程中的所有可执行文件。此外，可以定义其他的非默认目标文件列表用于其他目的，例如，当键入‘**make debug**’时会重新构建所有带有调试信息的文件。

### 3.11.2 本书中的makefile

使用本书第2卷的**ExtractCode.cpp**程序，本书中列出的所有代码会被自动地从本书的ASCII文本文件中抽取出来，并存放在相应章的子目录中。此外，**ExtractCode.cpp**程序会在每个子目录中创建一些**makefile**文件（具有不同的文件名），所以可以简单地进入子目录并输入**make -f mycompiler.makefile**（用你自己的编译器名来替换**mycompiler**，‘**-f**’标志说明跟在后面的是**makefile**文件）。最后**ExtractCode.cpp**程序在根目录中创建了一个“管理”**makefile**文件，在根目录中书中的文件已经被扩展，该**makefile**被传到各个子目录中且调用相应的**makefile**文件。这样发出一个**make**命令就能够编译本书中的所有代码，当编译器不能处理特别的文件（注意，与标准C++兼容的编译器能够编译本书中的所有文件）时，编译过程会停止。**make**的实现会随系统而异，因而在生成的**makefile**文件中仅仅使用了**make**的最基本的特征。

### 3.11.3 makefile的一个例子

正如提到的那样，代码提取工具**ExtractCode.cpp**自动地为每章产生**makefile**文件。因为这个原因，**makefile**并未放在书中每一章（所有的**makefile**文件都和源代码一起打包，可以从[www.BruceEckel.com](http://www.BruceEckel.com)下载）。

然而看一个**makefile**的例子是有意义的。以下是一个例子的简化版本，该例子由本书中的代码提取工具自动生成。可以在每个子目录中（它们有不同的名字，用‘**make -f**’调用）发现多个**makefile**文件。下面的例子是用于GNU C++的：

```

CPP = g++
OFLAG = -o
.SUFFIXES : .o .cpp .c
.cpp.o :
    $(CPP) $(CPPFLAGS) -c $<
.c.o :
    $(CPP) $(CPPFLAGS) -c $<

```

```

all: \
    Return \
    Declare \
    Ifthen \
    Guess \
    Guess2
# Rest of the files for this chapter not shown

Return: Return.o
    $(CPP) $(OFLAG)Return Return.o

Declare: Declare.o
    $(CPP) $(OFLAG)Declare Declare.o

Ifthen: Ifthen.o
    $(CPP) $(OFLAG)Ifthen Ifthen.o

Guess: Guess.o
    $(CPP) $(OFLAG)Guess Guess.o
Guess2: Guess2.o
    $(CPP) $(OFLAG)Guess2 Guess2.o

Return.o: Return.cpp
Declare.o: Declare.cpp
Ifthen.o: Ifthen.cpp
Guess.o: Guess.cpp
Guess2.o: Guess2.cpp

```

CPP宏被设置为编译器的名字。为了使用不同的编译器，可以编辑**makefile**文件，或者在命令行上修改宏的值，例如：

```
make CPP=cpp
```

注意，对于另外编译器，**ExtractCode.cpp**代码具有自动建立**makefile**的方案。

第二个宏**OFLAG**是一个标志，用于指定输出文件的名称。虽然许多编译器自动假定输出文件的名称与输入的文件名一致，但是还是有例外（如Linux/Unix编译器，它默认创建一个**a.out**的输出文件）。

可以看出本例有两条后缀规则，一条用于**cpp**文件，另一条用于**.c**文件（以防需要编译C代码）。默认的目标是**all**，对于目标的所有的行用反斜线符号表示继续，直到**Guess2**，它是目标列表中的最后一行，因此不再需要反斜线符。本章有许多文件，为简单起见，这里只列出了一些文件。

后缀规则管理从**cpp**文件创建目标文件（以**.o**作为扩展名），但是通常对创建可执行文件需要有显式说明的规则，因为一个可执行文件通常是通过连接许多不同的目标文件而产生的，而**make**程序不知道哪些是目标文件。同样，在某些情况（Linux/Unix）下，对于可执行文件并无标准扩展名，这种情况下，后缀规则将不能工作。所以，我们发现创建最终执行文件都显式说明了规则。

**makefile**采用最安全的路线，其中尽可能少地使用**make**特征，在宏中也使用了目标、依赖性和宏的最基本的**make**概念。这种方式实质上保证能与尽可能多的**make**程序共同工作。这可能会生成较大的**makefile**，但这不是很糟的事，因为它是通过**ExtractCode.cpp**自动产生的。

有许多本书中未使用的其他**make**特征，以及更新和更加灵活的**make**版本和变型，其中具有可以大量节约时间的高级快捷用法。本地文档可以对特定的**make**做更加详尽的描述，也可以从Oram和Talbot所著的《*Managing Projects with Make*》(O'Reilly, 1993)一书中学到关于**make**的更多的知识。如果有的编译器提供商不能支持**make**或者它使用非标准的**make**，可以从Internet上搜索GNU文档（有许多GNU文档）找到GNU **make**程序，这种程序实际上支持已经存在的所有平台。

### 3.12 小结

本章相当集中地浏览了C++语法的基本特征，许多特征是从C中继承过来的，同C是共有的（由此导致C++自夸与C向后兼容）。在这里虽然介绍了C++的某些特征，由于主要是针对熟悉编程的人，因此仅限于介绍C和C++的基本语法。如果读者已经是C程序员，那么除了C++的特征对读者多半是新的以外，还可能会发现一两点关于C的不熟悉的知识。

### 3.13 练习

部分练习题的答案可以在本书的电子文档“*Annotated Solution Guide for Thinking in C++*”中找到，只需支付很少的费用就可以从<http://www.BruceEckel.com>得到这个电子文档。

- 3-1 建立一个头文件（扩展名为‘.h’）。在该文件中，声明一组函数，具有可变参数，返回值包括**void**、**char**、**int**和**float**类型。建立一个包含上述头文件的**.cpp**文件，创建所有这些函数的定义。每个定义应该简单地输出函数名，参数列表，并返回类型以便知道它已经被调用。创建另外一个**.cpp**文件，它包含头文件且定义**int main()**，在其中调用已经定义的所有函数。编译和运行这个程序。
- 3-2 编写一个程序使用两重**for**循环和模运算符(**%**)去寻找和输出质数（只能被1和它本身整除的整数）。
- 3-3 编写一个程序，使用一个**while**循环从标准输入(**cin**)中把单词读入到**string**中。这是一个“无穷”**while**循环，可以使用**break**语句中断（和退出程序）。对于读入的每个单词，先用一系列的**if**语句把该单词“映射”为一个整数值，然后用该整数值作为一个**switch**语句的选择条件（这些操作并不意味着是良好的设计风格，这仅仅是为练习这些控制流程）。在每个**case**中，输出一些有意义的信息。判定哪些是“有趣”的单词以及这些单词的意义。同时判定哪个单词是程序结束的标志。用文件作为输入来测试该程序（如果想节省输入，这个文件将作为程序的源文件）。
- 3-4 修改**Menu.cpp**程序，使用**switch**语句代替**if**语句。
- 3-5 编写一个程序计算在“优先级”一节中的两个表达式的值。
- 3-6 修改**YourPets2.cpp**程序以使用不同的数据类型（**char**、**int**、**float**、**double**和这些类型的变型）。运行该程序并画出结果内存分布图。如果能在多种机器、操作系统或者编译器上运行该程序，用尽可能多的变化进行这个试验。
- 3-7 创建两个函数，一个接受一个**string\***参数，另一个接受一个**string&**参数。每个函数必须用它特有的方式去改变外部的**string**对象。在**main()**中，创建和初始化一个**string**对象，输出它，然后把它传给每个函数，输出结果。
- 3-8 编写一个使用所有三个图形字符（**trigraph**）的程序，看看你的编译器是否支持它们。

- 3-9 编译和运行**Static.cpp**程序。从代码中删除**static**关键词，再次编译和运行，解释发生的现象。
- 3-10 试编译**FileStatic.cpp**和**FileStatic2.cpp**程序并把它们连接起来。得到的错误消息的含义是什么？
- 3-11 修改**Boolean.cpp**程序，用**double**值代替**int**值。
- 3-12 修改**Boolean.cpp**和**Bitwise.cpp**程序，使用显式运算符（如果你的编译器与C++标准兼容，那么它会支持这些运算符）。
- 3-13 使用在**Rotation.cpp**程序中的函数去修改**Bitwise.cpp**程序。确保用这种方式能清楚地显示在旋转过程中的结果。
- 3-14 修改**Ifthen.cpp**程序，使用三重**if-else**运算符(**?:**)。
- 3-15 创建一个含有两个**string**对象和一个**int**对象的**struct**。使用**typedef**为该**struct**命名。创建**struct**的一个实例，初始化实例的三个值，然后输出它们。获得实例的地址，然后赋值给定义的**struct**类型的指针。改变实例的三个值，然后通过指针把它们打印出来。
- 3-16 编制一个使用颜色枚举类型的程序。创建一个**enum**类型的变量，然后用**for**循环输出与颜色名对应的数字。
- 3-17 用**Union.cpp**程序做一个试验，删除各种**union**元素，观察对**union**大小的影响。试给该**union**的一个元素赋值（属于某一类型），然后通过不同的元素（属于不同的类型）输出它的值，看看发生了什么情况。
- 3-18 编制一个程序，连续定义两个**int**数组。第二个数组的开始下标紧接第一个数组的结束下标。给两个数组赋值。打印出第二个数组观察由此引起的变化。再在两个数组定义之间定义一个**char**变量，重复上述操作。可以创建一个数组输出函数以简化程序。
- 3-19 修改**ArrayAddresses.cpp**程序，使之能处理**char**、**long**、**int**、**float**以及**double**类型数据。
- 3-20 运用**ArrayAddresses.cpp**程序中的技术，输出在**StructArray.cpp**程序中定义的**struct**的大小以及数组元素的地址。
- 3-21 创建一个**string**对象数组且对每一个元素赋一个字符串。用**for**循环输出该数组。
- 3-22 在**ArgsToInts.cpp**的基础上，编制两个新程序，它们各自使用**atol()**和**atof()**函数。
- 3-23 修改**PointerIncrement2.cpp**程序，其中用**union**代替**struct**。
- 3-24 修改**PointerArithmetic.cpp**程序，其中使用**long**和**long double**。
- 3-25 定义一个**float**变量。获得它的地址，把地址转化为**unsigned char**，赋值给一个**unsigned char**指针。使用指针和**[ ]**符号引用**float**变量中的下标，并用本章中定义的**printBinary()**函数输出该**float**的内存映像。（从0到**sizeof(float)**）。改变该**float**变量的值看看是否能推算出下一步的情况（**float**包含编码的数据）。
- 3-26 定义一个**int**数组。获得该数组的起始地址，使用**static\_cast**把它转化为**void\***。写一个带以下参数的函数：一个**void\***、一个数字（表明字节的数目）和一个值（表明每个字节需要设定的值）。该函数必须为特定范围内的每个字节设定特定的值。在这个**int**数组上试验函数。
- 3-27 建立一个**const double**类型数组和一个**volatile double**类型数组。通过引用每个数组的下标且用**const\_cast**把每个元素分别转换为**non-const**和**non-volatile**，然后对每个元素赋值。
- 3-28 建立一个函数，该函数接受一个指向**double**类型数组的指针和一个表明该数组大小的值。

该函数应该输出数组中的每个元素值。现在建立一个**double**类型的数组，且初始化每个元素的值为0，然后使用你的函数输出该数组。接着使用**reinterpret\_cast**关键字把数组的起始地址转化为**unsigned char\***，把每个元素值设置为1（提示：必须用**sizeof**运算符计算一个**double**类型变量包含的字节数）。现在使用你的数组输出函数输出结果。想想为什么每个元素值不设为1.0？

- 3-29 （带有挑战性）修改**FloatingAsBinary.cpp**程序以便能够以单独的二进制位组输出**double**类型数据。为实现目标，必须用自己的特殊代码（可以从**printBinary()**函数中衍生）去替换对**printBinary()**的调用，还必须查阅并理解自己的编译器的浮点数字节格式（这是具有挑战性的部分）。
- 3-30 创建**makefile**文件，可以把编译**YourPets1.cpp**和**YourPets2.cpp**程序（用你特定的编译器）以及执行这两个程序作为默认的目标，确保使用后缀规则。
- 3-31 修改**StringizingExpressions.cpp**程序，通过设置一个命令行标志，使得**P(A)**能用条件**#ifdef**与调试代码分离开。需要参考编译器文档，了解在命令行上怎样定义和取消定义预处理的值。
- 3-32 定义一个函数，该函数接受一个**double**型参数且返回一个**int**值。创建和初始化一个指向该函数的指针，通过这个指针调用这个函数。
- 3-33 声明一个函数，该函数接受一个**int**参数且返回指向另一个函数的指针，这个函数接受一个**char**变量且返回一个**float**值。
- 3-34 修改**FunctionTable.cpp**程序使每个函数返回一个**string**（而不是输出一个消息）以便在**main()**函数中输出。
- 3-35 为前面某个练习（自己选择）建立一个**makefile**文件，允许键入**make**以构建这个程序，并且键入**make debug**以构建带有调试信息的程序。



## 数据抽象

C++是一个能提高生产效率的工具。为什么我们要努力（不管我们试图做的转变多么容易，还是需要努力）使我们从已经熟悉且生产效率高的某种语言转到另一种新的语言上？而且使用这种新语言，我们会在确实掌握它之前的一段时间内降低生产效率。这是因为我们确信：通过使用新工具将会得到更大的好处。

用编程术语来讲，生产效率提高意味着较少的人能够在较少的时间内完成更复杂和更重要的程序。当然，选择语言时确实还有其他问题，例如运行效率（该语言的本质会引起运行速度减慢和代码臃肿吗？）、安全性（该语言能有助于确信我们的程序做我们计划的事情并具有很强的纠错能力吗？）、可维护性（该语言能帮助我们创建易理解、易修改和易扩展的代码吗？）。这些都是本书要介绍的重要因素。

简单地讲，提高生产效率，意味着本应当花费三个人一星期的程序，现在只需要花费一个人一两天的时间。这会涉及经济学的多层次问题。生产效率提高了，我们很高兴，因为我们正在建造的东西其功能将会更强；我们的客户（或老板）很高兴，因为产品生产又快，用人又少；我们的顾客很高兴，因为他们得到的产品更便宜。而大幅度提高生产效率的惟一办法就是使用其他人的代码，即是去使用库。

库只是他人已经写好的一些代码，按某种方式包装在一起。通常，最小的包是带有扩展名（如lib）的文件和向编译器声明库中有什么的一个或多个头文件。连接器知道如何在库文件中搜索和提取相应的已编译的代码。但是，这只是提供库的一种方法。在跨越多种体系结构的平台（例如Linux/Unix）上，通常，提供库的最明智的方法是使用源代码，这样它就能在新的目标机上被重新配置和编译。

所以，库大概是改进生产效率的最重要的方法。C++的主要设计目标之一就是使库使用起来更加容易。这种说法暗示，在C中使用库有一些难度。理解这个因素将使我们对于C++设计有一个初步的了解，并因而对如何使用它有更深入的认识。

### 4.1 一个袖珍C库

一个库通常以一组函数开始，但是，已经用过第三方C库的程序员知道，通常还有比行为、动作和函数更多的东西。有一些特性（颜色、重量、纹理、亮度），它们都由数据表示。在C语言中，当处理一组特性时，可以方便地把它放在一起，形成一个**struct**。特别是，如果我们想表示问题空间中的多个类似的东西时，可以对每件东西创建这个**struct**的一个变量。

这样，在大多数C库中都有一组**struct**和一组作用在这些**struct**之上的函数。现在看一个这样的例子。假设有一个编程工具，当创建时，它的表现像一个数组，但它的长度能在运行时建立。我称它为**CStash**。虽然它是用C++写的，但是它有C语言的风格：

```
//: C04:CLib.h
// Header file for a C-like library
// An array-like entity created at runtime
```



```

typedef struct CStashTag {
    int size;           // Size of each space
    int quantity;       // Number of storage spaces
    int next;           // Next empty space
    // Dynamically allocated array of bytes:
    unsigned char* storage;
} CStash;

void initialize(CStash* s, int size);
void cleanup(CStash* s);
int add(CStash* s, const void* element);
void* fetch(CStash* s, int index);
int count(CStash* s);
void inflate(CStash* s, int increase);
///<:~

```

像**CStashTag**这样的标签名一般用于需要在**struct**内部引用自身的情况。例如，如果创建一个链表（链表中的每个元素包含一个指向下一个元素的指针），这样就需要指向下一个**struct**变量的指针，所以需要一种方法，能辨别这个**struct**内部的指针的类型。在C库中，几乎总是可以在如上所示的每个**struct**体中看到**typedef**。这样做使得能把这个**struct**作为一个新类型处理，并且可以定义这个**struct**的变量，例如：

```
CStash A, B, C;
```

**storage**指针是一个**unsigned char\***。这是 C 编译器支持的最小的存储单位，尽管在某些机器上它可能与最大的一般大，这依赖于具体实现，但一般占一个字节长。人们可能认为，因为**CStash**被设计用于存放任何类型的变量，所以**void\***在这里应当更合适。然而，我们的目的并不是把它当做某个未知类型的块处理，而是作为连续的字节块。

这个实现文件的源代码（如果购买一个商品化的库，可能得到的只是编译好的**obj**或**lib**或**dll**等）如下：

```

//: C04:CLib.cpp {0}
// Implementation of example C-like library
// Declare structure and functions:
#include "CLib.h"
#include <iostream>
#include <cassert>
using namespace std;
// Quantity of elements to add
// when increasing storage:
const int increment = 100;

void initialize(CStash* s, int sz) {
    s->size = sz;
    s->quantity = 0;
    s->storage = 0;
    s->next = 0;
}

int add(CStash* s, const void* element) {
    if(s->next >= s->quantity) //Enough space left?
        inflate(s, increment);
    // Copy element into storage,
    // starting at next empty space:

```



```

    int startBytes = s->next * s->size;
    unsigned char* e = (unsigned char*)element;
    for(int i = 0; i < s->size; i++)
        s->storage[startBytes + i] = e[i];
    s->next++;
    return(s->next - 1); // Index number
}

void* fetch(CStash* s, int index) {
    // Check index boundaries:
    assert(0 <= index);
    if(index >= s->next)
        return 0; // To indicate the end
    // Produce pointer to desired element:
    return &(s->storage[index * s->size]);
}

int count(CStash* s) {
    return s->next; // Elements in CStash
}

void inflate(CStash* s, int increase) {
    assert(increase > 0);
    int newQuantity = s->quantity + increase;
    int newBytes = newQuantity * s->size;
    int oldBytes = s->quantity * s->size;
    unsigned char* b = new unsigned char[newBytes];
    for(int i = 0; i < oldBytes; i++)
        b[i] = s->storage[i]; // Copy old to new
    delete [] (s->storage); // Old storage
    s->storage = b; // Point to new memory
    s->quantity = newQuantity;
}

void cleanup(CStash* s) {
    if(s->storage != 0) {
        cout << "freeing storage" << endl;
        delete [] s->storage;
    }
}
} ///::~~

```

**initialize()**通过设置内部变量为适当的值。完成对**struct CStash**的必要设置。最初，设置**storage**指针为零，表示不分配初始存储。

**add()**函数在**CStash**的下一个可用位置上插入一个元素。首先，它检查是否有可用空间，如果没有，它就用后面介绍的**inflate()**函数扩展存储空间。

因为编译器并不知道存放的特定变量的类型（函数返回的都是**void\***），所以不能只做赋值，虽然这的确是很方便的事情。我们必须一个字节一个字节地拷贝这个变量，完成这项拷贝任务的最简单的方法是使用数组下标。典型的情况是，在**storage**中已经存放有数据字节，由**next**的值指明。为了从正确的字节偏移开始，**next**必须乘上每个元素的长度（按字节），产生**startBytes**，然后，参数**element**转换为一个**unsigned char\***，所以这就能一个字节接着一个字节地寻址，拷贝进可用的**storage**存储空间中。增加后的**next**指向下一个可用的存储块，**fetch()**能用指向这个数值存放点的“下标数”重新得到这个值。

**fetch()**首先看**index**是否越界,如果没有越界,返回所希望的变量地址,地址采用**index**参数计算。因为**index**指出了相对于**CStash**的偏移元素数,所以必须乘上每个单元拥有的字节数,产生按字节计算的偏移量。当此偏移用于计算使用数组下标的**storage**的下标时,得到的不是地址,而是处于这个地址上的字节。为了产生地址,必须使用地址操作符**&**。

对于有经验的C程序员,**count()**乍看上去可能有点奇怪,它好像是自找麻烦,做手工很容易做的事情。例如,如果有一个**struct CStash**,称为**intStash**,那么通过使用**intStash.next**查明它已经有多少个元素,这种方法似乎更直接,而不是去做**count(&intStash)**函数调用(它有更多的花费)。但是,如果想改变**CStash**的内部表示和计数计算的方法,那么这个函数调用接口就具有必要的灵活性。并且,很多程序员不会为找出库的“更好”的设计而操心。如果他们能着眼于**struct**和直接取**next**的值,那么就有可能不经允许而改变**next**。是不是能有一些方法使得库设计者能更好地控制像这样的问题呢?(是的,这是可预见的。)

#### 4.1.1 动态存储分配

我们不可能预先知道一个**CStash**需要的最大存储量是多少,所以从堆(*heap*)中分配由**storage**指向的内存。堆是很大的内存块,用以在运行时分配一些小的存储空间。在写程序时,如果还不知道所需内存的大小,就可以使用堆。这样,可以直到运行时才知道需要存放200个**Airplane**的空间,而不只是20个。在标准C中,动态内存分配函数包括**malloc()**、**calloc()**、**realloc()**和**free()**。然而,C++不是采用库调用方法,而是采用更高级的方法,即被集成进这个语言中的动态存储分配,使用关键字**new**和**delete**。

**inflate()**函数使用**new**为**CStash**得到更大的空间块。在这种情况下,只扩展内存而不缩小它,**assert()**保证不把负数传给**inflate()**作为**increase**的值。能够存储的新元素数(**inflate()**完成后)由计算**newQuantity**,再乘以每个元素的字节数得到**newBytes**,这是分配的字节数。因此,可以知道从旧的位置拷贝多少字节,**oldBytes**用旧的**quantity**计算。

实际的存储分配出现在**new**表达式中,它是包含**new**关键字的表达式:

```
new unsigned char[newBytes];
```

**new**表达式的一般形式是:

```
new Type;
```

其中**Type**表示希望在堆上分配的变量的类型。在这种情况下,我们希望一个长度为**newBytes**的**unsigned char**数组,这就是作为**Type**出现的变量。还可以分配简单类型的变量,例如**int**,表示为:

```
new int;
```

虽然很少这样做,但这可以使得形式一致。

**new**表达式返回指向所请求的准确类型对象的指针,因此,如果声称**new Type**,返回的是指向**Type**的指针。如果声称**new int**,返回指向一个**int**的指针。如果希望**new unsigned char**数组,返回的是指向这个数的第一个元素的指针。编译器确保把这个**new**表达式的返回值赋给一个正确类型的指针。

当然,任何时候申请内存都有可能失败,例如存储单元用完,正如我们看到的,C++有判断是否内存分配不成功的机制。

一旦分配了新内存块，旧内存块中的数据必须拷贝进这个新内存块，这又是通过数组下标完成的，在循环中一次拷贝一个字节。数据被拷贝以后，必须释放老的内存块，以便程序的其他部分在需要新内存块时使用。**delete**关键字是**new**的对应关键字，任何由**new**分配的内存块必须用**delete**释放（如果忘记了使用**delete**，这个内存块就不能用了，这称为内存泄漏（*memory leak*））。泄漏到一定程度，内存就耗尽了。另外，释放数组有特殊的语法形式，也就是必须提醒编译器，注意这是指向对象数组的指针，而不是仅仅指向一个对象的指针。该语法形式是在被释放的指针前面加一对空方括号：

```
delete []myArray;
```

一旦释放了旧的内存块，指向这个新内存块的指针就可以赋给**storage**指针，再调整数量，**inflate**就完成了任务。

注意，堆管理器是相当简单的，它给出一块内存，而当用**delete**释放时又把它收回。这里没有提供能压缩堆获得较大的空闲块的堆压缩内部工具。如果程序反复分配和释放堆存储，最终将会产生大量的空闲内存碎片，但却没有足够大的块能分配所需要的内存。堆压缩器使程序更复杂，因为要前后移动内存块，所以指针应保持正确的值。一些操作环境有内置的堆压缩器，但是，要求使用特殊的内存句柄（*handle*）（它能临时转换为指针，锁定内存后堆压缩器就不能移动它了）。

当编译时，如果在栈上创建一个变量，那么这个变量的存储单元由编译器自动开辟和释放。编译器准确地知道需要多少存储容量，根据这个变量的活动范围知道这个变量的生命期。而对动态内存分配，编译器不知道需要多少存储单元，不知道它们的生命期，不能自动清除。因此，程序员应负责用**delete**释放这块存储，**delete**告诉堆管理器，这个存储可以被下一次调用的**new**重用。在这个库里合理的方法是使用**cleanup()**函数，它做所有关闭的事情。

为了测试这个库，让我们创建两个**CStash**。第一个存放**int**，第二个存放由80个**char**组成的数组：

```
//: C04:CLibTest.cpp
//{L} CLib
// Test the C-like library
#include "CLib.h"
#include <fstream>
#include <iostream>
#include <string>
#include <cassert>
using namespace std;

int main() {
    // Define variables at the beginning
    // of the block, as in C:
    CStash intStash, stringStash;
    int i;
    char* cp;
    ifstream in;
    string line;
    const int bufsize = 80;
    // Now remember to initialize the variables:
    initialize(&intStash, sizeof(int));
    for(i = 0; i < 100; i++)
```



```

    add(&intStash, &i);
    for(i = 0; i < count(&intStash); i++)
        cout << "fetch(&intStash, " << i << ") = "
            << *(int*)fetch(&intStash, i)
            << endl;
    // Holds 80-character strings:
    initialize(&stringStash, sizeof(char)*bufsize);
    in.open("CLibTest.cpp");
    assert(in);
    while(getline(in, line))
        add(&stringStash, line.c_str());
    i = 0;
    while((cp = (char*)fetch(&stringStash,i++))!=0)
        cout << "fetch(&stringStash, " << i << ") = "
            << cp << endl;
    cleanup(&intStash);
    cleanup(&stringStash);
} ///:~

```

按照C语言的要求，所有的变量都在**main()**范围的开头定义。当然，必须在这个程序块的稍后通过调用**initialize()**对**CStash**初始化。C库的问题之一是必须向用户认真地说明初始化和清除函数的重要性，如果这些函数未被调用，就会出现许多问题。遗憾的是，用户不总是记得初始化和清除是必须的。他们只知道他们想完成什么，并不关心我们反复说的：“喂，等一等，您必须首先做这件事”。一些用户甚至认为初始化这些元素是自动完成的。在C中，的确没有机制能防止这种情况的发生（只有预示）。

**intStash**存放整型，**stringStash**存放字符数组。这些字符数组是通过打开源代码文件**CLibTest.cpp**和从中把这些行读到被称为**line**的**string**中形成的，然后使用成员函数**c\_str()**产生一个指向**line**字符的指针。

装载了这两个**Stash**之后，可以显示它们。**intStash**的打印用了一个**for**循环，用**count()**确定它的限度。**stringStash**的打印用一个**while**语句，如果**fetch()**返回零则表示打印越界，这时跳出循环。

还应当注意到下面的类型转换：

```
cp = (char*)fetch(&stringStash,i++)
```

这是因为C++有严格的类型检查，它不允许直接向其他类型赋**void\***（C允许）。

#### 4.1.2 有害的猜测

在考虑C库创建中的一般问题之前还应当了解一个更重要的问题。注意头文件**CLib.h**必须包含在所有涉及**CStash**的文件中，因为编译器不能正确地猜测这个结构像什么。然而，它能猜测一个函数像什么。这看上去像是一个特征，但实际上是C的一个主要缺陷。

虽然总是应当通过包含头文件声明函数，但是函数声明在C中不是基本的。调用没有声明的函数在C中是可以的（但是在C++中不可以）。一个好的编译器会告诫程序员应当首先声明函数，但是，按照C语言的标准，并不强迫这样。这是危险习惯，因为C编译器可能会假设，带有一个**int**参数的函数有包含**int**的参数表，尽管它实际上可能包含了一个**float**。正如我们将看到的，这会产生非常难发现的bug。

每个独立的C文件（带有扩展名**.c**的文件）是一个翻译单元（*translation unit*）。这就是说，

编译器在每个翻译单元上单独运行，这时它只知道这个单元。这样，由包含文件提供的任何信息都是相当重要的，因为它决定了编译器对程序的其他部分的理解。在头文件中的声明是特别重要的，因为在包含头文件的任何地方，编译器准确地知道做什么。例如，如果在头文件中有一个声明是 `void func(float)`，编译器就知道，如果用一个整型参数调用这个函数，应当把这个 `int` 转换为 `float`，作为传递参数 [这被称为提升(*promotion*)]。如果没有声明，C编译器简单地假设有一个 `func(int)` 存在，它就不做提升，错误数据就悄悄地传给了 `func()`。

对于每个翻译单元，编译器创建一个目标文件，用 `.o` 或者 `.obj`，或者其他类似的符号作为扩展名。这些目标文件，连同必要的启动代码，由连接器连接为可执行程序。在连接过程中，应当确定所有的外部引用。例如，在 `CLibTest.cpp` 中，声明和使用 `initialize()` 和 `fetch()` 这样的函数（这就是，告诉编译器它们像什么），但在其中未定义。它们在别处定义，即在 `CLib.cpp` 中。这样，在 `CLib.cpp` 中的调用是外部引用。当连接器将所有的对象文件放在一起时，它必须取未确定的外部引用，找出它们实际访问的地址。在可执行程序中用这些地址替换这些外部引用。

在C中，连接器所要查找的外部引用是一些简单的函数名字，通常在它们的前面加下划线。因此，所有的连接器都必须匹配调用处的函数名和在对象文件中的函数体。如果在某处我们调用一个函数 `func(int)`，而在某一目标文件中有 `func(float)` 的函数体，连接器将认为有 `_func` 在一处而且有 `_func` 在另一处，它认为这都对，在调用 `func()` 的地方，把 `int` 置入栈中，而 `func()` 函数体处认为 `float` 在栈中。如果这个函数只读这个值而不写，它不会破坏这个栈。事实上，如果它读取的这个 `float` 值可能刚好有某种意思，这是最坏的情况，因为这个bug很难找出。

## 4.2 哪儿出问题

我们通常有特别的适应能力，即使是对本不应该适应的事情。`CStash` 库的风格对于C程序员已经是常用的了，但是如果观察它一会儿，就会发现它是相当笨拙的。因为在使用它时，必须向这个库中的每一个函数传递这个结构的地址。而当读这些代码时，这种库机制会和函数调用的含义相混淆，当试图理解这些代码时也会引起混乱。

在C中，使用库的最大的障碍之一是名字冲突 (*name clashes*)。对于函数，C使用单个名字空间，当连接器查找一个函数名时，它在一个主表中查找，而且，当编译器编译一个单元时，它只能对带有指定名字的单个函数进行处理工作。

假设决定要从不同的厂商购买两个库，并且每一个库都有一个必须被初始化和清除的结构。两个厂商都认为 `initialize()` 和 `cleanup()` 是好名字。如果在某个处理单元中同时包含了这两个库文件，C编译器怎么办呢？幸好，标准C出错，报告声明函数有两个不同的参数表中类型不匹配。即便不把它们包含在同一个处理单元中，连接器也会有问题。好的连接器会发现这里有名字冲突，但有些编译器仅仅通过查找目标文件表，按照在连接表中给出的次序，取第一个找到的函数名（实际上，这可以看做是一种功能，因为可以用自己的版本替换一个库函数）。

无论哪种情况，都不允许使用包含具有同名函数的两个C库。为了解决这个问题，C库厂商常常会在它们的所有函数名前加上一个独特字符串。所以，`initialize()` 和 `cleanup()` 可能变为 `CStash_initialize()` 和 `CStash_cleanup()`。这是合乎逻辑的，因为它“修饰了”这个 `struct` 的名字，而该函数以这样的函数名对这个 `struct` 操作。

现在到了迈向C++第一步的时候。我们知道，**struct**内部的标识符不会与全局标识符冲突。而当一些函数在特定**struct**上运算时，为什么不把这一优点扩展到函数名上呢？也就是，为什么不把函数成为 **struct**的成员呢？

### 4.3 基本对象

C++的第一步正是这样，函数可以放在**struct**内部，作为“成员函数”。CStash的C版本翻译成C++的Stash后是：

```
//: C04:CppLib.h
// C-like library converted to C++

struct Stash {
    int size;           // Size of each space
    int quantity;       // Number of storage spaces
    int next;           // Next empty space
    // Dynamically allocated array of bytes:
    unsigned char* storage;
    // Functions!
    void initialize(int size);
    void cleanup();
    int add(const void* element);
    void* fetch(int index);
    int count();
    void inflate(int increase);
}; ///:~
```

首先，注意到这里没有**typedef**，而是要求程序员创建一个**typedef**。C++编译器把结构名转变为这个程序的新类型名（就像**int**、**char**、**float**和**double**是类型名一样）。

所有的数据成员与以前完全相同，但现在这些函数在**struct**的内部了。另外，注意到，对应于这个库中的C版本中第一个参数已经去掉了。在C++中，不是硬性传递这个结构的地址作为在这个结构上运算的所有函数的第一个参数，而是编译器秘密地做这件事。现在，这些函数的仅有的参数与它们所做的事情有关，而不与这些函数的运算机制有关。

认识到这些函数代码与在C库中的那些同样有效，是很重要的。参数的个数是相同的（虽然看不到这个结构地址被传进来，实际上它在这里），每个函数只有一个函数体。正因为如此，书写

```
Stash A, B, C;
```

并不意味着每个变量得到不同的**add()**函数。

那样产生的代码几乎和为C库写的一样。更有趣的是，这包括了“名字修饰”，在C中也许应当像**Stash\_initialize()**、**Stash\_cleanup()**等这样修饰。当函数在**struct**内时，编译器有效地做了同样的事情。因此，在**Stash**内部的**initialize()**将不会与任何其他结构中的 **initialize()** 相冲突，即便是与全局函数名**initialize()**，也不会冲突。大部分时间都不必为函数名字修饰而担心——而是使用未修饰的函数名。但有时还必须能够指出这个**initialize()**属于这个**struct Stash**而不属于任何其他的**struct**。特别是，当正在定义这个函数时，需要完全指定它是哪一个。为了完成这个指定任务，C++有一个新的运算符(**::**)，即作用域解析运算符（这样命名是因为名字现在能在不同的范围内：在全局范围内或在一个**struct**的范围内）。例如，如果希望



指定`initialize()`属于`Stash`，就写`Stash::initialize(int size)`。可以看到，在下面函数定义中是如何使用作用域运算符的：

```
//: C04:CppLib.cpp {0}
// C library converted to C++
// Declare structure and functions:
#include "CppLib.h"
#include <iostream>
#include <cassert>
using namespace std;
// Quantity of elements to add
// when increasing storage:
const int increment = 100;

void Stash::initialize(int sz) {
    size = sz;
    quantity = 0;
    storage = 0;
    next = 0;
}

int Stash::add(const void* element) {
    if(next >= quantity) // Enough space left?
        inflate(increment);
    // Copy element into storage,
    // starting at next empty space:
    int startBytes = next * size;
    unsigned char* e = (unsigned char*)element;
    for(int i = 0; i < size; i++)
        storage[startBytes + i] = e[i];
    next++;
    return(next - 1); // Index number
}

void* Stash::fetch(int index) {
    // Check index boundaries:
    assert(0 <= index);
    if(index >= next)
        return 0; // To indicate the end
    // Produce pointer to desired element:
    return &(storage[index * size]);
}

int Stash::count() {
    return next; // Number of elements in CStash
}

void Stash::inflate(int increase) {
    assert(increase > 0);
    int newQuantity = quantity + increase;
    int newBytes = newQuantity * size;
    int oldBytes = quantity * size;
    unsigned char* b = new unsigned char[newBytes];
    for(int i = 0; i < oldBytes; i++)
        b[i] = storage[i]; // Copy old to new
    delete []storage; // Old storage
```



```

    storage = b; // Point to new memory
    quantity = newQuantity;
}

void Stash::cleanup() {
    if(storage != 0) {
        cout << "freeing storage" << endl;
        delete []storage;
    }
} //::~~

```

在C和C++之间有以下不同：首先，头文件中的声明是由编译器要求的。在C++中，不能调用未事先声明的函数，否则编译器将报告一个出错信息。这是确保这些函数调用在被调用点和被定义点之间一致的重要方法。通过强迫在调用函数之前必须声明它，C++ 编译器可以保证我们用包含这个头文件的方式完成这个声明。如果在这个函数被定义的地方还包含有同样的头文件，则编译器将作一些检查以保证在这个头文件中的声明和这个定义匹配。这意味着，这个头文件变成了函数声明的有效的仓库，并且保证这些函数在项目中的所有处理单元中使用一致。

当然，全局函数仍然可以在定义和使用它的每个地方用手工方式声明（这是很乏味的，以致变得不太可能）。然而，必须在定义和使用之前声明结构，而最习惯放置结构定义的位置是在头文件中，除非有意把它藏在代码文件中。

可以看到，除了作用域和来自这个库的 C 版本的第一个参数不再是显式的这一事实以外，所有这些成员函数实际上都与C版本中的一样。当然，这个参数仍然存在，因为这个函数必须工作在一个特定的**struct**变量上。但是，在成员函数内部，成员照常使用。这样，不写**s->size = sz**，而写**size = sz**。这就消除了多余的**s->**，它对我们所做的任何事情不能添加任何含义。当然，C++ 编译器必须为我们做这些事情。实际上，它取“秘密”的第一个参数（也就是先前用手工传递的这个结构的地址），并且当提到**struct**的数据成员的任何时候，应用成员选择器。这意味着，当在另一个**struct**的成员函数中时，通过简单地给出成员的名字，就可以使用任何成员（包括其他成员函数）。编译器在找出这个名字的全局版本之前先在局部结构的名字中搜索。这个性能意味着不仅代码更容易写，而且更容易阅读。

但是，如果因为某种原因，我们希望能够处理这个结构的地址，情况会怎么样呢？在这个库的C版本中，这是很容易的，因为每个函数的第一个参数是叫做**s**的一个**CStash\***。在C++ 中，事情是更一致的。这里有一个特殊的关键字，称为**this**，它产生这个**struct**的地址。它等价于这个库的C版本的‘**s**’。所以可以用下面语句恢复成C风格。

```
this->size = Size;
```

对这种书写形式进行编译所产生的代码是完全一样的，因此不需要像这样的方式使用**this**。有时，我们会看到有人在代码的各处都明显地使用**this->**，但是，这不能对代码增加任何意义。通常，不经常用**this**，而只是需要时才使用（稍后，本书中将有一些使用**this**的例子）。

最后需要提到，在C中，可以赋**void\***给任何指针，例如：

```

int i = 10;
void* vp = &i; // OK in both C and C++
int* ip = vp; // Only acceptable in C

```

而且编译器能够通过。但在 C++ 中，这个语句是不允许的。为什么呢？因为 C 对类型信息不挑剔，所以它允许未明确类型的指针赋给一个明确类型的指针。而 C++ 则不同。类型在 C++ 中是严格的，当类型信息有任何违例时，编译器就不允许。这一点一直是很重要的，而对于 C++ 尤其重要，因为在 **struct** 中有成员函数。如果能够在 C++ 中向 **struct** 传递指针而不被阻止，那么就能最终调用对于 **struct** 逻辑上并不存在的成员函数。这是防止灾难的一个实际的办法。因此，C++ 允许将任何类型的指针赋给 **void\***（这是 **void\*** 的最初的意图，它需要足够大，以存放任何类型的指针），但不允许将 **void** 指针赋给任何其他类型的指针。一个类型转换总是需要告诉读者和编译器，我们实际上要把它作为目标类型处理。

这就带来了一个有趣的问题，C++ 的最重要的目的之一是能编译尽可能多的已存在的 C 代码，以便能容易地向这个新语言过渡。然而，这并不意味着 C 允许的任何代码都能自动地被 C++ 接受。有一些 C 编译器允许的东西是危险的和易出错的（本书中还会看到它们）。C++ 编译器对于这些情况产生警告和出错信息，其优点远大于缺点。实际上，在 C 中有许多我们知道有错误只是不能找出它的情况，但是一旦用 C++ 重编译这个程序，编译器就能指出这些问题。在 C 中，我们常常发现能使程序通过编译，然后我们必须再花力气使它工作。在 C++ 中，常常是，程序编译正确了，它也就能工作了。这是因为该语言对类型要求更严格的缘故。

在下面的测试程序中，可以看到 **Stash** 的 C++ 版本所使用的另一些东西。

```
//: C04:CppLibTest.cpp
//{L} CppLib
// Test of C++ library
#include "CppLib.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
    Stash intStash;
    intStash.initialize(sizeof(int));
    for(int i = 0; i < 100; i++)
        intStash.add(&i);
    for(int j = 0; j < intStash.count(); j++)
        cout << "intStash.fetch(" << j << ") = "
              << *(int*)intStash.fetch(j)
              << endl;
    // Holds 80-character strings:
    Stash stringStash;
    const int bufsize = 80;
    stringStash.initialize(sizeof(char) * bufsize);
    ifstream in("CppLibTest.cpp");
    assure(in, "CppLibTest.cpp");
    string line;
    while(getline(in, line))
        stringStash.add(line.c_str());
    int k = 0;
    char* cp;
    while((cp = (char*)stringStash.fetch(k++)) != 0)
        cout << "stringStash.fetch(" << k << ") = "
```

```

        << cp << endl;
    intStash.cleanup();
    stringStash.cleanup();
} ///:~

```

我们可以注意到，变量是实时（on the fly）定义的（第3章介绍过）。也就是说，它们能在作用域内的任何点上定义，而不是像C语言限制的那样，只能在作用域的开头部分。

这段代码和ClibTest.cpp相似，但在调用成员函数时，在函数名字之前使用成员选择运算符“.”。这是一个传统的文法，它模仿了结构数据成员的使用。它们的不同在于这里是函数成员，有一个参数表。

当然，该编译器实际产生的调用，看上去更像原来的C库函数。如果考虑名字修饰和this传递，C++ 函数调用 `intStash.initialize(sizeof(int), 100)`就和`Stash_initialize(&intStash, sizeof(int), 100)`一样了。如果想知道在内部所进行的工作，可以回忆最早的C++ 编译器**cfront**，它由AT&T开发，它输出的是C代码，然后再由C 编译器编译。这个方法意味着**cfront** 能使C++很快地移植到有C 编译器的机器上，有助于快速地传播 C++ 编译器技术。正因为这个C++编译器必须产生C，所以我们知道必然有方法用C语言描述C++文法（某些编译器仍然允许产生C代码）。

ClibTest.cpp的另一个改变是引入**require.h**头文件，这是为这本书创造的头文件，用来完成比**assert()**更复杂的错误检查任务。它包含了几个函数，其中一个就是在这里为了检查文件而使用的**assure()**。它检查这个文件是否已经成功地打开了，如果没有，它就报告一个标准错误，告诉这个文件不能打开并且退出程序（这样，它就需要文件名作为第二个参数）。**require.h**函数在全书中都会用到，特别是为了保证命令行参数的个数正确和保证文件确实打开了。**require.h**取代了不断重复的和分散进行的检查出错代码，并且提供非常有用的出错信息。这些函数将在本书的后面解释。

## 4.4 什么是对象

我们已经看到了一个最初的例子，现在回过头来看一些术语。把函数放进结构中是从C到C++中的根本改变，这引起我们将结构作为新概念去思考。在C中，**struct**是数据的凝聚，它将数据捆绑在一起，使得我们可以将它们看做一个包。但这除了能使编程方便之外，别无其他。对这些结构进行操作的函数可以在别处。然而将函数也放在这个包内，结构就变成了新的造物了，它既能描写属性（就像C **struct** 能做的一样），又能描述行为，这就形成了对象的概念。对象是一个独立的捆绑的实体，有自己的记忆和活动。

在C++中，对象就是变量，它的最纯正的定义是“一块存储区”（更明确的说法是，“对象必须有惟一的标识”，在C++中是一个惟一的地址）。它是一块空间，在这里能存放数据，而且还隐含着有对这些数据进行处理的操作。

不幸的是，对于各种语言，当涉及这些术语时，并不完全一致，尽管它们是可以接受的。我们有时还会遇到关于面向对象语言是什么的争论，虽然到目前为止看起来已经相当调和了。有一些语言是基于对象的（*object-based*），意味着它们有像C++ 的结构加函数这样的对象，正如已经看到的。然而，这只是到达面向对象语言历程中的一部分，停留在把函数捆绑在数据结构内部的语言是基于对象的，而不是面向对象的。

## 4.5 抽象数据类型

将数据连同函数捆绑在一起的能力可以用于创建新的数据类型。这常常被称为封装 (*encapsulation*)<sup>①</sup>。一个已存在的数据类型可能有几块数据封装在一起,例如 **float**, 有一个指数, 一个尾数和一个符号位。我们能够告诉它做事情: 与另一个 **float** 或 **int** 相加, 等等。它有属性和行为。

**Stash** 的定义创建了一个新数据类型, 可以 **add()**、**fetch()** 和 **inflate()**。由说明 **Stash s** 创建一个 **Stash** 就像由说明 **float f** 创建一个 **float** 一样。一个 **Stash** 也有属性和行为, 甚至它的活动就像一个实数——一个内建的数据类型。称 **Stash** 为抽象数据类型 (*abstract data type*), 也许这是因为它能允许从问题空间抽象概念到解空间。另外, C++ 编译器把它看做一个新的数据类型, 如果说一个函数需要一个 **Stash**, 编译器就确保传递了一个 **Stash** 给这个函数。对抽象数据类型 [有时称为用户定义类型 (*user-defined type*)] 的类型检查就像对内建类型的类型检查一样严格。

然而, 我们会看到在对象上执行操作的方法有所不同。**object.member Function(arglist)** 是对一个对象“调用一个成员函数”。而在面向对象的用法中, 也称之为“向一个对象发送消息”。这样, 对于 **Stash s**, 语句 **s.add(&i)** “发送消息给 **s**”, 也就是说, “将它与自己 **add()**”。事实上, 面向对象编程可以总结为一句话, “向对象发送消息”。实际上, 需要做的所有事情就是创建一束对象并且给它们发送消息。当然, 技巧是勾画出对象和消息是什么, 但如果完成了这些, 用 C++ 的实现就直截了当了。

## 4.6 对象细节

在研讨会上经常提出的一个问题是“对象应当多大和它应当像什么”。回答是“和 C 的 **struct** 一样”。事实上, 对于 C **struct** (不带有 C++ 的改进), 由 C 编译器产生的代码和由 C++ 编译器产生的完全相同, 这可以使那些在代码中离不开结构的安排和大小细节的程序员放心, 并且由于某种原因, 他们直接访问结构的字节而不是使用标识符。(具体取决于不可移植的结构体的特定大小和布局。)

一个 **struct** 的大小是它的所有成员大小的和。有时, 当一个 **struct** 被编译器处理时, 会增加额外的字节以使得边界整齐, 这主要是为了提高执行效率。在第 15 章中, 将会看到如何在结构中增加“秘密”指针, 但现在不必关心这些。

用 **sizeof** 运算符可以确定 **struct** 的长度。这里有一个小例子:

```
//: C04:Sizeof.cpp
// Sizes of structs
#include "CLib.h"
#include "CppLib.h"
#include <iostream>
using namespace std;

struct A {
    int i[100];
};

struct B {
    void f();
};
```

① 这个词会引起争论, 一些人采用此处的定义, 还有一些人用它描述访问权限控制 (在第 5 章讨论)。

```
};

void B::f() {}

int main() {
    cout << "sizeof struct A = " << sizeof(A)
          << " bytes" << endl;
    cout << "sizeof struct B = " << sizeof(B)
          << " bytes" << endl;
    cout << "sizeof CStash in C = "
          << sizeof(CStash) << " bytes" << endl;
    cout << "sizeof Stash in C++ = "
          << sizeof(Stash) << " bytes" << endl;
} ///:~
```

在我的机器上（你的机器可能有不同的结果），第一个打印语句产生的结果是200，因为每个 **int** 占两个字节。**struct B** 是奇异的，因为它没有数据成员的 **struct**。在 C 中，这是不合法的，但在 C++ 中，以这种选择方式创建一个 **struct**，惟一的目的就是划定函数名的范围，所以这是允许的。尽管如此，由第二个打印语句产生的结果是一个有点奇怪的非零值。在该语言的较早的版本中，这个长度是零，但是，当创建这样的对象时出现了笨拙的情况：它们与紧跟着它们创建的对象有相同的地址，没有区别。对象的基本规则之一是每个对象必须有一个惟一的地址，因此，无数据成员的结构总应当有最小的非零长度。

最后两个 **sizeof** 语句表明在 C++ 中的结构长度与 C 中等价版本的长度相同。C++ 尽力不增加任何不必要的开销。

## 4.7 头文件形式

当创建了一个包含有成员函数的 **struct** 时，也就创建了一个新数据类型。一般情况，希望这个类型对于我们和其他人都容易使用。另外，还希望将接口（声明）和实现（成员函数的定义）隔离开来，使得实现能在不需要重新编译整个系统的情况下可以改变。最后，将这个新类型的声明放到头文件中。

当我第一次学习用 C 编程时，头文件对我是神秘的。许多有关 C 语言的书似乎不强调它，并且编译器也并不强调函数声明，所以它在大部分时间内似乎是可要可不要的，除非要声明结构时。在 C++ 中，头文件的使用变得非常明显。它们对于很容易的程序开发实际上是强制，在它们中放入非常特殊的信息：声明。头文件告诉编译器在我们的库中哪些是可用的。即便程序员只拥有头文件和对象文件或库文件，他也能用这个库。因为对于 **cpp** 文件能够不要源代码而使用库。头文件是存放接口规范的地方。

虽然编译器不强迫这样做，但是，用 C++ 建造大项目的最好的方法是采用库，收集相关的函数到同一个对象模块或库中，并且使用同一个头文件存放所有这些函数的声明。在 C++ 中这是必须的；在 C 中，可以把所有的函数都放进 C 库中，但是在 C++ 中，由抽象数据类型确定库中的函数，这些函数通过它们共同访问一个 **struct** 中数据而联系起来。任何成员函数必须在 **struct** 声明中声明，不能把它放在其他地方。在 C 中，鼓励使用函数库，而在 C++ 中，这是一项制度。

### 4.7.1 头文件的重要性

当使用库函数时，C 允许不用头文件，而是简单地随手声明这个函数。过去，人们有时候这样做是为了通过避免打开和包含这个文件而略微提高编译器的速度（这对于现代编译器一

般不是问题)。例如, 这里有C函数`printf()`的一个非常简单的声明 (来自`<stdio.h>`):

```
printf(...);
```

省略号表示可变的参数表<sup>⊖</sup>, 说明`printf()`有一些参数, 每个参数有类型, 但省略了它们。这样, 无论什么参数都接受。使用这样的声明, 就中止了对参数的检查。

这种习惯会引起问题, 如果随手声明了一个函数, 在一个文件中可能会留下错误。因为编译器只看到在这个文件中的手工声明, 它可能会适应错误。然后这个程序会被正确地连接, 但是这样使用函数, 一个文件将会出现错误。这是很难发现的错误, 而使用头文件就可以很容易地避免这种情况。

如果将所有的函数声明都放在一个头文件中, 并且将这个头文件包含在使用这些函数和定义这些函数的任何文件中, 就能确保在整个系统中声明的一致性。通过将这个头文件包含在定义文件中, 还可以确保声明和定义匹配。

在C++中, 如果在一个头文件中声明了一个`struct`, 我们在`struct`的任何地方和定义这个`struct`成员函数的任何地方必须包含这个头文件。如果不经声明就调用常规函数, 调用或定义成员函数, C++编译器会给出错误消息。通过强制正确地使用头文件, 语言保证库中的一致性, 并通过在各处强制使用相同的接口, 可以减少程序错误。

头文件是我们和我们的库的用户之间的合约。这份合约描述了我们的数据结构, 为函数调用规定了参数和返回值。它说, “这里是对我的库做什么的描述。” 用户需要其中一些信息以开发应用程序, 编译器需要所有这些以生成正确的代码。这个`struct`的用户简单地包含这个头文件, 创建这个`struct`的对象 (实例), 连接到对象模块或库 (也就是被编译的代码) 中。

通过要求我们在`struct`和函数之前声明所有这些`struct`和函数, 在定义成员函数之前声明这些成员函数, 编译器强制履行这个合约。这样, 就强制我们在头文件中放置这些声明, 强制将这个头文件包含在定义成员函数的文件中和使用这些函数的文件中。因为描述库的单个头文件被包含在整个系统各处, 所以编译器能确保一致性和防止错误。

我们必须认识到为了正确地组织代码和编写有效的头文件, 需要考虑几个具体问题。第一个问题涉及应当放什么到头文件中。基本的原则是“只限于声明”, 即只限于对编译器的信息, 不涉及通过生成代码或创建变量而分配存储的任何信息。这是因为头文件一般会包含在项目的几个翻译单元中, 如果一个标识符在多于一处被分配存储, 那么连接器就报告多次定义错误 (这是C++的一次定义规则: 可以对事物声明任意多次, 但是对于每个事物只能实际定义一次)。

这条规则不是呆板的。如果在头文件中定义了一个“文件静态”变量 (仅在一个文件内可视的变量), 那么在整个项目中会有该数据的多个实例, 但连接器不会冲突<sup>⊖</sup>。基本上, 我们不希望做任何会引起连接时歧义性的事情。

#### 4.7.2 多次声明问题

头文件的第二个问题是: 如果把一个`struct`声明放在一个头文件中, 就有可能在一个编译程序中多次包含这个头文件。输入输出流就是一个很好的例子。每次一个`struct`做I/O都可能

⊖ 写一个带有可变参数列表的函数定义, 必须应用`varargs`, 虽然在C++中应避免这样使用。`varargs`的应用细节请参照C手册。

⊖ 然而, 在标准C++文件中, `static`是一个不予推荐的特征。



包含一个输入输出流文件。如果我们正在开发的**cpp**文件使用多种**struct**（典型的是每种包含一个头文件），这样就有多次包含**<iostream>**和重声明输入输出流的危险。

编译器认为重声明结构（包括**struct**和**class**）是一个错误，因为它还允许对不同的类型使用相同的名字。为了防止多次头文件包含引起的错误，需要在头文件中用预处理器建立一些智能功能（标准C++头文件中**<iostream>**等已经具有这样的“智能”）。

C和C++都允许重声明函数，只要两个声明匹配即可，但是两者都不允许重声明结构。在C++中，这条规则是特别重要的，因为如果编译器允许重声明一个结构而且这两个声明不同，那么应当使用哪一个声明呢？

重声明在C++中出现了问题，因为每个数据类型（带函数的结构）一般有它自己的头文件，如果想创造另一个数据类型（它使用第一个数据类型），则我们必须将第一个数据类型的头文件包含在这另一个数据类型中。在我们项目的任何**cpp**文件中，很可能包含几个已经包含了这个相同的头文件的文件。在一次编译过程中，编译器可能会多次看到这个相同的头文件。除非特别处理，否则编译器将发现结构的重声明，并报告编译时错误。为了解决这个问题，需要知道更多的预处理器的知识。

#### 4.7.3 预处理器指示**#define**、**#ifdef**和**#endif**

预处理器指示**#define**可以用来创建编译时标记。你有两种选择：你可以简单地告诉预处理器这个标记被定义，但不指定特定的值：

```
#define FLAG
```

或者给它一个值（这是典型的定义常数的C方法）：

```
#define PI 3.14159
```

无论哪种情况，预处理器都能测试该标记，检查它是否已经被定义：

```
#ifdef FLAG
```

这将得到一个真值，**#ifdef**后面的代码将包含在发送给编译器的包中。当预处理器遇到语句

```
#endif
```

或

```
#endif // FLAG
```

时包含终止。

在同一行中，**#endif**之后无注释是不合规定的，尽管一些编译器可以接受这样的行。**#ifdef/#endif**对可以相互嵌套。

**#define**的反意是**#undef**（“un-define”的简写），它将使得使用相同变量的**#ifdef**语句得到假值。**#undef**还引起预处理器停止使用宏。**#ifdef**的反意是**#ifndef**，如果标记还没有定义，它得到真值（这是在头文件中使用的指示）。

在C预处理器中还有其他有用的特性，因此我们还应当检查我们文档中的全部设置。

#### 4.7.4 头文件的标准

对于包含结构的每个头文件，应当首先检查这个头文件是否已经包含在特定的**cpp**文件中

了。这需要通过测试预处理器的标记来检查。如果这个标记没有设置，这个文件没有包含，则应当设置它（所以这个结构不会被重声明），并声明这个结构。如果这个标记已经设置，则表明这个类型已经声明了，所以应当忽略这段声明它的代码。下面显示头文件的样子：

```
#ifndef HEADER_FLAG
#define HEADER_FLAG
// Type declaration here...
#endif // HEADER_FLAG
```

正如已经看到的，头文件第一次被包含，这个头文件的内容（包括类型声明）将被包含在预处理器中。对于在单个编译单元中的所有后续的包含，该类型声明被忽略。`HEADER_FLAG`可以是任何惟一的名称，但沿用的可靠标准是大写这个头文件的名称并且用下划线替换句点（但是前面的下划线是为系统名保留的）。例如：

```
//: C04:Simple.h
// Simple header that prevents re-definition
#ifndef SIMPLE_H
#define SIMPLE_H

struct Simple {
    int i,j,k;
    initialize() { i = j = k = 0; }
};
#endif // SIMPLE_H ///:~
```

虽然`#endif`之后的`SIMPLE_H`是注释，并且预处理器忽略它，但它对于文档是有用的。防止多次包含的这些预处理器语句常常称为包含守卫(include guard)。

#### 4.7.5 头文件中的名字空间

我们将会注意到，在这本书的几乎所有`cpp`文件中都有使用指令（*using directive*）描述，通常的形式如下：

```
using namespace std;
```

因为`std`是环绕整个标准C++库的名字空间，所以这个特定的使用指令允许不用限定方式使用标准C++库中的名字。但是，在头文件中是决不会看到使用指令的（至少，不在一个作用域之外）。原因是，这样的使用指令去除了对这个特定名字空间的保护，并且这个结果一直持续到当前编译单元结束。如果将一个使用指令放在一个头文件中（在一个范围之外），这就意味着“名字空间保护”将在包含这个头文件的任何文件中消失，这些文件常常是其他的头文件。这样，如果将使用指令放在头文件中，将很容易最终实际上在各处“关闭”名字空间，因此不能体现名字空间的好处。

简言之，不要在头文件中放置使用指令。

#### 4.7.6 在项目中使用头文件

当用C++建立项目时，我们通常要汇集大量不同的类型（带有相关函数的数据结构）。一般将每个类型或一组相关类型的声明放在一个单独的头文件中，然后在一个处理单元中定义这个类型的函数。当使用这个类型时必须包含这个头文件，执行正确的声明。

有时这个模式会在本书中使用，但是，更常见的情况是例子很小，所以结构声明、函数定义和 **main()** 函数可以出现在同一个文件中。然而，应当记住，你想要实际使用的是隔离的文件和头文件。

## 4.8 嵌套结构

在全局名字空间之外为数据和函数取名字的好处可以扩展到结构中。我们可以将一个结构嵌套在另一个结构中，这就可以将相关联的元素放在一起。声明文法是我们所期望的形式，就像在下面结构中可以看到的那样，这个结构用简单链表方式实现了一个下推栈 (push-down stack)，所以它绝不会越出内存。

```
//: C04:Stack.h
// Nested struct in linked list
#ifndef STACK_H
#define STACK_H

struct Stack {
    struct Link {
        void* data;
        Link* next;
        void initialize(void* dat, Link* nxt);
    }* head;
    void initialize();
    void push(void* dat);
    void* peek();
    void* pop();
    void cleanup();
};
#endif // STACK_H ///:~
```

这个嵌套 **struct** 称为 **Link**，它包括一个指向这个表中的下一个**Link**的指针和一个指向存放在 **Link** 中的数据的指针。如果 **next** 指针是零，这就意味着到了表尾。

注意：**head** 指针紧接在 **struct Link** 声明之后定义，而不是单独定义 **Link\* head**。这是来自C语言的一种文法，但它强调在结构声明之后的分号的重要性，分号表明这个结构类型用逗号分开的定义表结束（通常这个定义表是空的）。

正如到目前为止所有描述的结构一样，嵌套结构有它自己的 **initialize()** 函数，以便确保正确的初始化。**Stack** 既有 **initialize()** 函数又有 **cleanup()** 函数，此外还有 **push()** 函数，它取一个指向希望存放的数据（假设已经分配在堆中）的指针；还有 **pop()** 函数，它返回栈顶的 **data** 指针并去除栈顶元素。（注意，当 **pop()** 一个元素时，我们有责任销毁由 **data** 所指的对象）。**peek()** 函数也从栈顶返回 **data** 指针，但是它在栈 (**Stack**) 中保留这个栈顶元素。

下面是一些成员函数的定义：

```
//: C04:Stack.cpp {0}
// Linked list with nesting
#include "Stack.h"
#include "../require.h"
using namespace std;
void
Stack::Link::initialize(void* dat, Link* nxt) {
    data = dat;
```

```

    next = nxt;
}

void Stack::initialize() { head = 0; }

void Stack::push(void* dat) {
    Link* newLink = new Link;
    newLink->initialize(dat, head);
    head = newLink;
}

void* Stack::peek() {
    require(head != 0, "Stack empty");
    return head->data;
}

void* Stack::pop() {
    if(head == 0) return 0;
    void* result = head->data;
    Link* oldHead = head;
    head = head->next;
    delete oldHead;
    return result;
}

void Stack::cleanup() {
    require(head == 0, "Stack not empty");
} //::~~

```

第一个定义特别有趣，因为它表明如何去定义一个嵌套结构的成员。只需要使用一个额外的作用域解析层，说明外围 **struct** 的名字。**Stack::Link::initialize()** 函数取参数并把参数赋给它的成员们。

**Stack::initialize()** 函数置 **head** 为零，使得这个对象知道它有一个空表。

**Stack::push()** 取参数，也就是一个指向希望用的变量的指针，并且把这个指针推入 **Stack**。首先，使用 **new** 为 **Link** 分配空间，它将插入栈顶。然后调用 **Link** 的 **initialize()** 函数对这个 **Link** 的成员赋相应的值。注意，给 **next** 指针赋当前的 **head**，而给 **head** 赋新的 **Link** 指针。这就有效地将 **Link** 推向这个表的顶部了。

**Stack::pop()** 取出当前在该 **Stack** 顶部的 **data** 指针，然后向下移 **head** 指针，删除该 **Stack** 的旧的栈顶元素，最后返回这个取出的指针。当 **pop()** 取出了最后的元素后，**head** 再次变为零，意味着 **Stack** 为空。

实际上，**Stack::cleanup()** 不做任何清除工作，而是确立一项硬性的策略，“你（即使用这个 **Stack** 对象的客户程序员）负责弹出这个 **Stack** 的所有元素并且删除它们。”**require()** 指出：如果 **Stack** 非空，就产生一个编程错误。

为什么 **Stack** 的析构函数不能对客户程序员不做 **pop()** 的所有对象负责呢？问题是，**Stack** 存放的是 **void** 指针，而且在第13章中我们将了解对 **void\*** 调用 **delete** 不能正确地清除内容。“谁对内存负责”这个主题不是一个简单的问题，在后面章节中将会看到相关的内容。

下面是一个测试 **Stack** 的例子：

```
//: C04:StackTest.cpp
```

```

//{L} Stack
//{T} StackTest.cpp
// Test of nested linked list
#include "Stack.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char* argv[]) {
    requireArgs(argc, 1); // File name is argument
    ifstream in(argv[1]);
    assure(in, argv[1]);
    Stack textlines;
    textlines.initialize();
    string line;
    // Read file and store lines in the Stack:
    while(getline(in, line))
        textlines.push(new string(line));
    // Pop the lines from the Stack and print them:
    string* s;
    while((s = (string*)textlines.pop()) != 0) {
        cout << *s << endl;
        delete s;
    }
    textlines.cleanup();
} ///:~

```

这个例子非常类似于前面一个例子，但是它把来自文件的行（作为**string**指针）存放到**Stack**中，然后弹出它们，这会使这个文件被逆序打印出来。注意**pop()**成员函数返回一个**void\***，并且必须在被用之前转换回**string\***。间接引用指针以便打印**string**。

当填充**textlines**时，通过建立**new string(line)**为每个**Push()**“复制”**line**的内容。从新表达式返回的值是指向这个新创建的**string**的指针，并且从**line**复制信息。如果简单地传递**line**地址给**push()**，最终用相同的地址充填**Stack**，所有的指针都指向**line**。在本书的后面，我们将学习更多的“复制”过程。

文件名取自命令行。为了保证在命令行上有足够的参数，我们看**require.h**头文件中的第二个函数**requireArgs()**，它比较**argc**与期望的参数个数，如果没有足够的参数，打印相应的错误信息并退出程序。

#### 4.8.1 全局作用域解析

编译器默认选择的名称（“最接近”的名称）可能不是我们要用的名称，作用域解析运算符可以避免这种情况。例如，假设有一个结构，它的局域标识符为**a**，但是我们希望在成员函数内选用全局标识符**a**。这时，编译器将默认选择局域的另一个标识符，因而必须告诉编译器应该选择哪个标识符。当你要用作用域解析运算符指定一个全局名字时，在运算符前面不添加任何东西。下面是一个显示变量和函数的全局作用域解析的例子：

```

//: C04:Scoperes.cpp
// Global scope resolution
int a;

```

```

void f() {}

struct S {
    int a;
    void f();
};

void S::f() {
    ::f(); // Would be recursive otherwise!
    ::a++; // Select the global a
    a--;   // The a at struct scope
}

int main() { S s; f(); } ///:~

```

如果在 `S::f()` 中没有作用域解析运算符，编译器会默认地选择成员函数的 `f()` 和 `a`。

## 4.9 小结

在本章中，我们学习了使用C++的基本方法，也就是在结构的内部放入函数。结构的这种新类型称为抽象数据类型 (*abstract data type*)，用这种结构创建的变量称为这个类型的对象 (*object*) 或实例 (*instance*)。调用对象的成员函数称为向这个对象发消息 (*sending a message*)。在面向对象的程序设计中的主要动作就是向对象发消息。

虽然将数据和函数捆绑在一起有很大好处，并使得库更容易使用（因为这可以通过隐藏名字防止名字冲突），但是还有大量的工作可以使C++编程更安全。在第5章中，我们将学习如何保护 `struct` 的一些成员，以使得只有我们能对它们进行操作。这就在“什么是结构的用户可以改动的”和“什么只是程序员可以改动的”之间建立了明确的界线。

## 4.10 练习

部分练习题的答案可以在本书的电子文档“*Annotated Solution Guide for Thinking in C++*”中找到，只需支付很少的费用就可以从<http://www.BruceEckel.com>得到这个电子文档。

- 4-1 在标准C库中，函数 `puts()` 能显示字符数组到控制台上（所以能写 `puts("hello")`）。试写一个C语言程序，这个程序使用 `puts()`，但不包含 `<stdio.h>`，也不声明这个函数。用C编译器编译这个程序。（有些C++编译器并不与它们的C编译器分开；在这种情况下，可能需要使用一个强制C编译的命令行标记。）然后再用C++编译器对它编译，注意它们之间的区别。
- 4-2 创建一个 `struct` 声明，它有单个成员函数，然后为这个成员函数创建定义。创建这个新数据类型的对象，再调用这个成员函数。
- 4-3 改变练习2的答案，使得 `struct` 在合适的“防护”头文件中声明，同时，它的定义在一个 `cpp` 文件中，`main()` 在另一个文件中。
- 4-4 创建一个 `struct`，它有一个 `int` 数据成员，再创建两个全局函数，每个函数都接受一个指向该 `struct` 的指针。第一个函数有第二个 `int` 参数，并设置这个 `struct` 的 `int` 为它的参数值，第二个函数显示来自这个 `struct` 的 `int`。测试这两个函数。
- 4-5 重写练习4，将两个函数改为这个 `struct` 的成员函数，再次测试。
- 4-6 创建一个类，它使用 `this` 关键字（冗余地）执行数据成员选择和成员函数调用。（`this` 表

示当前对象的地址)。

- 4-7 让**Stash**存放**double**，存入25个**double**值，然后把它们显示到控制台上。
- 4-8 用**Stack**重写练习7。
- 4-9 创建一个文件，包含以**int**为参数的函数**f()**，用**<stdio.h>**中的**printf()**函数将参数**int**的值显示到控制台上，即写**printf("%d\n",i)**，这里*i*是希望显示的**int**。创建另外一个单独的文件，它包含**main()**，在该文件中声明**f()**接受**float**参数。从**main()**中调用**f()**。尝试用C++编译器编译和连接这个程序，看看会发生什么事情。再用C编译器编译和连接这个程序，观察运行时会发生什么事情。解释这里的行为。
- 4-10 发现如何由你的C编译器和C++编译器产生汇编语言。用C写一个函数和用C++写一个带有一个成员函数的**struct**。由每一个编译器产生汇编语言，找出由你的C函数和C++成员函数产生的函数名，这样，你能看到什么样的名字修饰出现在编译器内部。
- 4-11 写一个**main()**中有条件编译代码的程序，使得当预处理器的值被定义时打印一条消息，而不被定义时则打印另外一条消息。编译这一代码段在程序中有**#define**的试验代码，然后找出你的编译器在命令行上定义预处理器的方法，对它进行试验。
- 4-12 写一个程序，它带有参数总是为假（零）的**assert()**，当运行时看发生什么现象。现在用**#define NDEBUG**编译它，再次运行它，看有什么不同。
- 4-13 创建一个抽象数据类型，它表示录像带租赁店中的录像带，试考虑在录像带租赁管理系统中为使录像带（**Video**）类型运作良好而必须的所有数据与运算。包含一个能显示录像带**Video**信息的**print()**的成员函数。
- 4-14 创建一个**Stack**对象，能存放练习13中的**Video**对象。创建几个**Video**对象，把它们存放在**Stack**中，然后用**Video::print()**显示它们。
- 4-15 写一个程序，使用**sizeof**打印出你的编译器的所有基本数据类型的长度。
- 4-16 修改**Stash**，使用**vector<char>**作为它的底层数据结构。
- 4-17 使用**new**动态创建下面类型的存储块：**int**、**long**、一个能存放100个**char**的数组、一个能存放100个**float**的数组。打印它们的地址，然后用**delete**释放这些存储。
- 4-18 写一个带有**char\***参数的函数。用**new**动态申请一个**char**数组，长度与传给这个函数的**char**数组同。使用数组下标，从参数中拷贝字符到这个动态申请的数组中（不要忘记**null**终结符）并且返回拷贝的指针。在**main()**中，通过传递静态引用字符数组，测试这个函数。然后取这个结果，再传回这个函数。打印这两个字符串和这两个指针，这样我们可以看到它们是不同的存储。使用**delete**，清除所有的动态存储。
- 4-19 显示在一个结构中声明另一个结构的例子（嵌套结构），声明这两个**struct**的数据成员，声明和定义这两个**struct**的成员函数。写一个**main()**，测试这两个新类型。
- 4-20 结构有多大？写一段代码，打印几个结构的长度。创建几个只有数据成员的结构和几个既有数据成员又有函数成员的结构，然后创建一个完全没有成员的结构。打印出所有这些结构的长度。解释产生完全没有成员的结构的结果的原因。
- 4-21 C++自动创建**struct**的**typedef**的等价物，正如在本章中看到的。对于枚举和联合类型也是如此。写一个小程序来证明这一点。
- 4-22 创建一个存放**Stash**的**Stack**，每个**Stash**存放来自输入文件的5行。使用**new**创建**Stash**，读文件进入**Stack**，然后从这个**Stack**中提取并按原来的形式打印出来。



- 4-23 修改练习22，使得创建一个**struct**，它封装几个**Stash**的**Stack**。用户只能通过成员函数添加和得到一行，在这个覆盖下，**struct**使用**Stash**的**Stack**。
- 4-24 创建一个**struct**，它存放一个**int**和一个指向相同**struct**的另一个实例的指针。写一个函数，它能取这些**struct**的地址，并且能取一个表示被创建的表的长度的**int**。这个函数产生这些**struct**的一个完整链（链表），链表的头指针是这个函数的参数，**struct**中的指针指向下一个**struct**。用**new**产生一些新**struct**，将计数（对象数目）放在这个**int**中，对这个链表的最后一个**struct**的指针栏置零值，表示链表结束。写第二个函数，它取这个链表的头指针，并且向后移动到最后，打印出每个**struct**的指针值和**int**值。
- 4-25 重复练习24，但是将这些函数放在一个**struct**内部，而不是用“原始”的**struct**和函数。



## 隐藏实现

一个典型的C语言库通常包含一个**struct**和一些作用在这个**struct**上面的相关函数。迄今为止，我们已经看到C++怎样处理那些在概念上相关联的函数，并使它们在语义上真正关联起来，具体做法是：

把函数的声明放在一个**struct**的范围之内，改变这些函数的调用方法，在调用过程中不再把结构地址作为第一个参数传递，并增加一个新的数据类型到程序中（这样就不必为**struct**标记创建一个**typedef**之类的声明）。

这样做带来很多方便——有助于组织代码，使程序易于编写和阅读。然而，在使得C++中的库比以前更容易的同时，还存在一些其他问题，特别是在安全与控制方面。本章重点讨论结构中的边界问题。

### 5.1 设置限制

在任何关系中，设立相关各方都遵从的边界是很重要的。一旦建立了一个库，我们就与该库的客户程序员（*client programmer*）建立了一种关系，客户程序员需要用我们的库来编写应用程序或建立另外的库。

在C语言中，**struct**同其他数据结构一样，没有任何规则，客户程序员可以在**struct**中做他们想做的任何事情，没有什么途径来强制任何特殊的行为。比如，即使已经看到了第4章中提到的**initialize()**函数和**cleanup()**函数的重要性，但客户程序员有权决定是否调用它们（我们将在下一章看到更好的方法）。再比如，我们可能不愿意让客户程序员去直接操纵**struct**中的某些成员，但在C语言中没有任何方法可以阻止客户程序员这样做。一切都是暴露无遗的。

需要控制对结构成员的访问有两个理由：一是让客户程序员远离一些他们不需要使用的工具，这些工具对数据类型内部的处理来说是必需的，但对客户程序员解决特定问题的接口却不是必须的。这实际上是为客户程序员提供了方便，因为他们可以很容易地知道，对他们来说什么是重要的，什么是可以忽略的。

访问控制的理由之二是允许库的设计者改变**struct**的内部实现，而不必担心会对客户程序员产生影响。在上一章的**Stack**例子中，我们想以大块的方式来分配存储空间以提高速度，而不是在每次增加元素时调用**malloc()**函数来重新分配内存。如果这些库的接口部分与实现部分是清楚地分离并保护的，那么就能达到上述目的并且只需要让客户程序员重新连接一遍就可以了。

### 5.2 C++的访问控制

C++语言引进了三个新的关键字，用于在结构中设置边界：**public**、**private**和**protected**。它们的用法和含义从字面上就能理解。这些访问说明符（*access specifier*）只在结构声明中，它们可以改变跟在它们之后的所有声明的边界。无论什么时候使用访问说明符，后面必须加

一个冒号。

**public**意味着在其后声明的所有成员可以被所有的人访问。**public**成员就如同一般的**struct**成员。比如，下面的**struct**声明是相同的：

```
//: C05:Public.cpp
// Public is just like C's struct

struct A {
    int i;
    char j;
    float f;
    void func();
};

void A::func() {}

struct B {
public:
    int i;
    char j;
    float f;
    void func();
};

void B::func() {}

int main() {
    A a; B b;
    a.i = b.i = 1;
    a.j = b.j = 'c';
    a.f = b.f = 3.14159;
    a.func();
    b.func();
} ///:~
```

相对地，**private**关键字则意味着，除了该类型的创建者和类的内部成员函数之外，任何人都不能访问。**private**在设计者与客户端程序员之间筑起了一道墙。如果有人试图访问一个私有成员，就会产生一个编译错误。在上面的例子中，我们可以让**struct B**中的部分数据成员隐藏起来，只有我们自己能访问它们：

```
//: C05:Private.cpp
// Setting the boundary

struct B {
private:
    char j;
    float f;
public:
    int i;
    void func();
};

void B::func() {
    i = 0;
    j = '0';
    f = 0.0;
}
```



```
};

int main() {
    B b;
    b.i = 1;    // OK, public
    //! b.j = '1'; // Illegal, private
    //! b.f = 1.0; // Illegal, private
} ///:~
```

虽然函数**func()**可以访问**B**的所有成员（因为**func()**是**B**的成员，所以自动获得访问的权限），但一般的全局函数如**main()**却不能访问，当然其他结构的成员函数同样也不能访问。只有那些在结构声明（“合约”）中明确声明的函数才能访问这些**private**成员。

对访问说明符的顺序没有特别的要求，它们可以出现不止一次，可以影响在它们之后和下一个访问说明符之前声明的所有成员。

### 5.2.1 **protected**说明符

最后一种访问说明符是**protected**。**protected**与**private**基本相似，只有一点不同：继承的结构可以访问**protected**成员，但不能访问**private**成员。这个问题要到第14章才讨论继承，那时会更清楚。现在可以把这两种说明符等同看待。

## 5.3 友元

如果程序员想允许显式地不属于当前结构的一个成员函数访问当前结构中的数据，那该怎么办呢？他可以在该结构内部声明这个函数为**friend**（友元）。注意，一个**friend**必须在一个结构内声明，这一点很重要，因为程序员（和编译器）必须能读取这个结构的声明以理解这个数据类型的大小、行为等方面的所有规则。有一条规则在任何关系中都很重要，那就是“谁可以访问我的私有实现部分”。

类控制着哪些代码可以访问它的成员。如果不是一个**friend**的话，程序员没有办法从类外“破门而入”，他不能声明一个新类，然后说“嘿，我是类**Bob**的朋友（友元）”，不能指望这样就可以访问类**Bob**的**private**成员和**protected**成员。

程序员可以把一个全局函数声明为**friend**，也可以把另一个结构中的成员函数甚至整个结构都声明为**friend**，请看下面的例子：

```
//: C05:Friend.cpp
// Friend allows special access

// Declaration (incomplete type specification):
struct X;

struct Y {
    void f(X*);
};

struct X { // Definition
private:
    int i;
public:
    void initialize();
```



```

    friend void g(X*, int); // Global friend
    friend void Y::f(X*); // Struct member friend
    friend struct Z; // Entire struct is a friend
    friend void h();
};

void X::initialize() {
    i = 0;
}

void g(X* x, int i) {
    x->i = i;
}

void Y::f(X* x) {
    x->i = 47;
}

struct Z {
private:
    int j;
public:
    void initialize();
    void g(X* x);
};

void Z::initialize() {
    j = 99;
}

void Z::g(X* x) {
    x->i += j;
}

void h() {
    X x;
    x.i = 100; // Direct data manipulation
}

int main() {
    X x;
    Z z;
    z.g(&x);
} //::~~

```

**struct Y**有一个成员函数`f()`，它将修改**X**类型的对象。这里有一个难题，因为C++的编译器要求在引用任一变量之前必须先声明，所以**struct Y**必须在它的成员`Y::f(X*)`被声明为**struct X**的一个友元之前声明，但要声明`Y::f(X*)`，又必须先声明**struct X**。

解决的办法：注意到`Y::f(X*)`引用了一个**X**对象的地址 (*address*)。这一点很关键，因为编译器知道如何传递一个地址，这一地址具有固定的大小，而不管被传递的是什么对象，即使它还没有完全知道这种对象类型大小。然而，如果试图传递整个对象，编译器就必须知道**X**的全部定义以确定它的大小以及如何传递，这就使得程序员无法去声明一个类似于`Y::g(X)`的函数。

通过传递**X**的地址，编译器允许程序员在声明`Y::f(X*)`之前做一个**X**的不完全的类型说明

(*incomplete type specification*)。这一点是用如下的声明时完成的:

```
struct X;
```

该声明仅仅是告诉编译器, 有一个叫X的**struct**, 所以当它被引用时, 只要不涉及名字以外的其他信息, 就不会产生错误。

这样, 在**struct X**中, 就可以成功地声明**Y :: f(X\*)**为一个**friend**函数, 如果程序员在编译器获得Y的全部说明信息之前声明它, 就会产生一条错误, 这种安全措施保证了数据的一致性, 同时减少了错误的出现。

再来看看其他两个**friend**函数, 第一个声明将一个全局函数**g()**作为一个**friend**, 但**g()**在这之前并没有在全局范围内作过声明, 这表明**friend**可以在声明函数的同时又将它作为**struct**的友元。这种扩展声明对整个结构同样有效:

```
friend struct Z;
```

是Z的一个不完全的类型说明, 并把整个结构都当做一个**friend**。

### 5.3.1 嵌套友元

嵌套的结构并不能自动获得访问**private**成员的权限。要获得访问私有成员的权限, 必须遵守特定的规则: 首先声明(而不定义)一个嵌套的结构, 然后声明它是全局范围使用的一个**friend**, 最后定义这个结构。结构的定义必须与**friend**声明分开, 否则编译器将不把它看做成员。请看下面的例子:

```
//: C05:NestFriend.cpp
// Nested friends
#include <iostream>
#include <cstring> // memset()
using namespace std;
const int sz = 20;

struct Holder {
private:
    int a[sz];
public:
    void initialize();
    struct Pointer;
    friend Pointer;
    struct Pointer {
private:
        Holder* h;
        int* p;
public:
        void initialize(Holder* h);
        // Move around in the array:
        void next();
        void previous();
        void top();
        void end();
        // Access values:
        int read();
        void set(int i);
    };
};
```



```

};

void Holder::initialize() {
    memset(a, 0, sz * sizeof(int));
}

void Holder::Pointer::initialize(Holder* rv) {
    h = rv;
    p = rv->a;
}

void Holder::Pointer::next() {
    if(p < &(h->a[sz - 1])) p++;
}

void Holder::Pointer::previous() {
    if(p > &(h->a[0])) p--;
}

void Holder::Pointer::top() {
    p = &(h->a[0]);
}

void Holder::Pointer::end() {
    p = &(h->a[sz - 1]);
}

int Holder::Pointer::read() {
    return *p;
}

void Holder::Pointer::set(int i) {
    *p = i;
}

int main() {
    Holder h;
    Holder::Pointer hp, hp2;
    int i;

    h.initialize();
    hp.initialize(&h);
    hp2.initialize(&h);
    for(i = 0; i < sz; i++) {
        hp.set(i);
        hp.next();
    }
    hp.top();
    hp2.end();
    for(i = 0; i < sz; i++) {
        cout << "hp = " << hp.read()
              << ", hp2 = " << hp2.read() << endl;
        hp.next();
        hp2.previous();
    }
} ///:~

```





一旦**Pointer**被声明，它就可以通过下面语句来获得访问**Holder**的私有成员的权限：

```
friend Pointer;
```

**struct Holder**包含一个**int**数组和一个**Pointer**，可以通过**Pointer**来访问这些整数。因为**Pointer**与**Holder**紧密联系，所以有必要将它作为结构**Holder**中的一个成员。但是，又因为**Pointer**是同**Holder**分开的，所以程序员可以在函数**main()**中定义它们的多个实例，然后用它们来选择数组的不同部分。由于**Pointer**是一个结构而不是C语言中原始意义上的指针，因此程序员可以保证它总是安全地指向**Holder**的内部。

使用标准C语言库函数 **memset()** (在<cstring>中)可以使上面的程序变得容易。它把起始于某一特定地址的内存(该内存作为第一个参数)从起始地址直至其后的**n** (**n**作为第三个参数)个字节的所有内存都设置成同一个特定的值 (该值作为第二个参数)。当然，程序员可以使用一个简单的循环来反复设置需要使用的所有内存，而且，**memset()**是可用的，经过了很好的测试不太可能引入错误，而且比起手工编码来更有效。

### 5.3.2 它是纯面向对象的吗

这种类定义提供了有关权限的信息，通过查看该类可以知道哪些函数可以改变该类的私有部分。如果一个函数被声明为**friend**，就意味着它不是这个类的成员函数，却可以修改该类的私有成员，而且必须被列在该类的定义当中，因此可以认为它是一个特权函数。

C++不是完全的面向对象语言，而只是一个混合产品。增加**friend**关键字就是为了用来解决一些实际问题。这也说明了这种语言是不纯的。毕竟C++语言的设计目的是实用，而不是追求理想的抽象。

## 5.4 对象布局

第4章讲述了为C编译器而写的一个**struct**，然后一字不动地用C++编译器进行编译。这里分析**struct**的对象布局，也就是，各个变量放在分配给对象的内存的什么位置？如果C++编译器改变了C **struct**中的布局，那么在任意C语言代码中使用**struct**中变量的位置信息在C++中就会出错。

当开始使用访问说明符时，我们就已经完全进入了C++的领地，情况开始有所改变。在一个特定的“访问块”（被访问说明符限定的一组声明）内，这些变量在内存中肯定是连续存放的，这和在C语言中一样，然而这些“访问块”本身可以不按声明的顺序在对象中出现。虽然编译器通常都是按访问块出现的顺序给它们分配内存，但并不是一定要这样，因为特定机器的体系结构和操作环境可对**private**成员和**protected**成员提供明确的支持，将其放在特定的内存位置上。C++语言的访问说明符并不想限制这种长处。

访问说明符是结构的一部分，它们并不影响从这个结构创建的对象。程序开始运行之前，所有的访问说明信息都消失了。访问说明信息通常是在编译期间消失的。在程序运行期间，对象变成了一个存储区域，别无他物，因此，如果有人真的想破坏这些规则并且直接访问内存中的数据，就如在C中所做的那样，那么C++并不能防止他做这种不明智的事，它只是提供给人们一个更容易、更方便的方法。

一般说来，在程序员编写程序时，依赖特定实现的任何东西都是不合适的。如确有必要，这些特定实现部分应封装在一个结构之内，这样当环境改变时，只需修改一个地方就行了。

## 5.5 类

访问控制通常是指实现细节的隐藏 (*implementation hiding*)。将函数包含到一个结构内 (常称为封装<sup>⊖</sup>) 来产生一种带数据和操作的数据类型, 由访问控制在该数据类型之内确定边界。这样做的原因有两个: 首先是决定哪些客户程序员可以用, 哪些客户程序员不能用。我们可以建立结构内部的机制, 而不必担心客户程序员会把内部的数据机制当做他们可使用的接口的一部分来访问。

这就直接导出了第二个原因, 那就是将具体实现与接口分离开来。如果该结构被用在一系列的程序中, 而客户程序员只能对**public**的接口发送消息, 这样就可以改变所有声明为**private**的成员而不必去修改客户程序员的代码。

同时采用封装和访问控制可以防止一些情况的发生, 而这在C语言的**struct**类型中是做不到的。现在我们已经处在面向对象编程的世界中, 在这里, 结构就是由对象组成的类, 就像人们可以描述由鱼或鸟组成的类一样, 任何属于该类的对象都将共享这些特征和行为。也就是说, 结构的声明已经变成该类型的所有对象看起来像什么以及将如何行动的描述。

在最初的面向对象编程语言Simula-67中, 关键字**class**被用来描述一个新的数据类型。这显然启发了Stroustrup在C++中选用同样的关键字, 以强调这是整个语言的关键所在。新的数据类型并非只是C中的**struct**加上函数, 这当然需要用一个新的关键字。

然而在C++中使用的**class**逐渐变成了一个非必要的关键字。它和**struct**的每个方面都是一样的, 除了**class**中的成员默认为**private**, 而**struct**中的成员默认为**public**。下面有两个结构, 它们将产生相同的结果。

```
//: C05:Class.cpp
// Similarity of struct and class

struct A {
private:
    int i, j, k;
public:
    int f();
    void g();
};

int A::f() {
    return i + j + k;
}

void A::g() {
    i = j = k = 0;
}

// Identical results are produced with:

class B {
    int i, j, k;
public:
    int f();
    void g();
};
```

⊖ 正如前面说明的一样, 访问控制有时被认为是一种封装。



```
};

int B::f() {
    return i + j + k;
}

void B::g() {
    i = j = k = 0;
}

int main() {
    A a;
    B b;
    a.f(); a.g();
    b.f(); b.g();
} ///:~
```

在C++中，**class**是面向对象编程的基本概念，它是一个关键字，但本书将不再用粗体字来表示——由于总要用到“class”，都标出来会令人厌烦。类带来的变化是如此重要，因此我怀疑Stroustrup偏向于将**struct**重新定义，但考虑到对C的向后兼容性而没有这样做。

许多人喜欢用一种更像**struct**的风格去创建一个类，因为可以通过不顾及类的“默认为**private**”的行为，而使用首选为**public**的原则。

```
class X {
public:
    void interface_function();
private:
    void private_function();
    int internal_representation;
};
```

之所以这样做，是因为这样可以让读者首先更清楚地看到他们所关心的成员，然后可以忽略所有声明为**private**的成员。事实上，所有其他成员都必须在类中声明的惟一原因是让编译器知道对象有多大，以便为它们分配合适的存储空间，并保证它们的一致性。

但本书中的示例仍采用首先声明**private**成员的格式，如下例：

```
class X {
    void private_function();
    int internal_representation;
public:
    void interface_function();
};
```

有些人甚至不厌其烦地在他们的私有成员名字前加上私有标志：

```
class Y {
public:
    void f();
private:
    int mX; // "Self-decorated" name
};
```

因为**mX**已经隐藏于**Y**的范围内，所以**m**（用它来指示成员）并不是必需的。然而在一个有许多全局变量的项目中（虽然极力想避免使用全局变量，但它们仍不可避免地的一些项目中出

现), 这种命名有助于在一个成员函数的定义体内识别出哪些是全局变量, 哪些是成员变量。

### 5.5.1 用访问控制来修改Stash

现在把第4章的例子用类及访问控制来改写一下。请注意客户程序员的接口部分现在已经很清楚地区分开了, 完全不用担心客户程序员会偶然地访问到他们不该访问的内容了。

```
//: C05:Stash.h
// Converted to use access control
#ifndef STASH_H
#define STASH_H

class Stash {
    int size;          // Size of each space
    int quantity;      // Number of storage spaces
    int next;          // Next empty space
    // Dynamically allocated array of bytes:
    unsigned char* storage;
    void inflate(int increase);
public:
    void initialize(int size);
    void cleanup();
    int add(void* element);
    void* fetch(int index);
    int count();
};
#endif // STASH_H ///:~
```

**inflate()**函数声明为**private**, 因为它只被**add()**函数调用, 所以它属于内部实现部分, 不属于接口部分。这就意味着以后可以调整这些实现的细节, 使用不同的系统来管理内存。

在此例中, 除了包含文件的名字之外, 只有上面的头文件需要更改, 实现文件和测试文件是相同的。

### 5.5.2 用访问控制来修改Stack

对于第二个例子, 我们把**Stack**改写成一个类。现在嵌套的数据结构是**private**。这样做的好处是可以确保客户程序员既看不到, 也不依赖于**Stack**的内部表示:

```
//: C05:Stack2.h
// Nested structs via linked list
#ifndef STACK2_H
#define STACK2_H

class Stack {
    struct Link {
        void* data;
        Link* next;
        void initialize(void* dat, Link* nxt);
    }* head;
public:
    void initialize();
    void push(void* dat);
    void* peek();
};
```



```

    void* pop();
    void cleanup();
};
#endif // STACK2_H ///:~

```

与上例一样，实现部分不需要改动，这里不再赘述。测试部分也一样，惟一改动了的地方是类的接口部分的健壮性。访问控制的真正价值体现在开发阶段中的防止越界。事实上，只有编译器知道类成员的保护级别，与成员关联的这些访问控制信息并没有被传递给连接器。所有的访问保护检查都是由编译器来完成的，在运行期间不再检查。

注意面向客户程序员的接口部分现在是一个压入式堆栈。它是用一个链表结构实现的，但可以换成其他的形式，而不会影响客户程序员处理问题，更重要的是，不需要改动客户程序员的代码。

## 5.6 句柄类

C++中的访问控制允许将实现部分与接口部分分开，但实现部分的隐藏是不完全的。编译器仍然必须知道一个对象的所有部分的声明，以便正确地创建和管理它。可以想象一种只需声明一个对象的公共接口部分的编程语言，它将私有的实现部分隐藏起来。但C++要尽可能多地在编译期间作静态类型检查。这意味着尽早捕获错误，也意味着程序具有更高的效率。然而包含私有实现部分会带来两个影响：一是即使客户程序员不能轻易地访问私有实现部分，但可以看到它；二是造成一些不必要的重复编译。

### 5.6.1 隐藏实现

有些项目不可让最终客户程序员看到其实现部分。例如可能在一个库的头文件中显示一些策略信息，但公司不想让这些被竞争对手获得。我们可能在从事一个安全性很重要的系统——比如一个加密算法——我们不想在文件中暴露任何线索，以防有人破译代码。或许我们把库放在了一个“有敌意”的环境中，在那里程序员会不顾一切地用指针和类型转换来访问我们的私有成员。在所有这些情况下，就有必要把一个编译好的实际结构放在实现文件中，而不是让其暴露在头文件中。

### 5.6.2 减少重复编译

在我们的编程环境中，当一个文件被修改，或它所依赖的头文件被修改时，项目管理员需要重复编译该文件。这意味着程序员无论何时修改了一个类，无论修改的是公共的接口部分，还是私有成员的声明部分，他都必须再次编译包含头文件的所有文件。这就是通常所说的易碎的基类问题 (*fragile base-class problem*)。对于一个大的项目而言，在开发初期这可能非常难以处理，因为内部实现部分可能需要经常改动。如果这个项目非常大，用于编译的时间过多可能妨碍项目的快速转型。

解决这个问题的技术有时称为句柄类 (*handle class*) 或称为“Cheshire cat”<sup>①</sup>。有关实现的任何东西都消失了，只剩一个单指针“smile”。该指针指向一个结构，该结构的定义与其

① 这个名字应该归功于John Carolan，他是C++语言早期的先驱之一，当然，还有Lewis Carroll（作家，《爱丽丝奇遇记》的作者——编辑注）。可以把该技术看做本书第2卷中论述的一种“桥” (bridge) 设计模式。

所有的成员函数的定义一同出现在实现文件中。这样，只要接口部分不改变，头文件就不需变动。而实现部分可以按需要任意更改，完成后只需要对实现文件进行重新编译，然后重新连接到项目中。

这里有一个说明这一技术的简单例子。头文件中只包含公共的接口和一个单指针，该单指针指向一个没有完全定义的类。

```
//: C05:Handle.h
// Handle classes
#ifndef HANDLE_H
#define HANDLE_H

class Handle {
    struct Cheshire; // Class declaration only
    Cheshire* smile;
public:
    void initialize();
    void cleanup();
    int read();
    void change(int);
};
#endif // HANDLE_H ///:~
```

这是所有客户程序员都能看到的。下面这行

```
struct Cheshire;
```

是一个不完整的类型说明 (*incomplete type specification*) 或类声明 (*class declaration*) [类定义 (*class definition*) 包含类的主体]。它告诉编译器，**Cheshire** 是一个结构名，但没有提供有关该**struct**的任何细节。这些信息对产生一个指向**struct**的指针来说已经足够了。但在提供了一个结构的主体部分之前不能创建一个对象。在这种技术里，包含具体实现的结构体被隐藏在实现文件中。

```
//: C05:Handle.cpp {0}
// Handle implementation
#include "Handle.h"
#include "../require.h"

// Define Handle's implementation:
struct Handle::Cheshire {
    int i;
};

void Handle::initialize() {
    smile = new Cheshire;
    smile->i = 0;
}

void Handle::cleanup() {
    delete smile;
}

int Handle::read() {
    return smile->i;
}
```



```
void Handle::change(int x) {
    smile->i = x;
} ///:~
```

**Cheshire** 是一个嵌套结构，所以它必须用作用域符定义：

```
struct Handle::Cheshire {
```

在 **Handle::initialize()** 中，为 **Cheshire** 结构分配存储空间，在 **Handle::cleanup()** 中，释放这些存储空间。这些内存被用来代替类的所有 **private** 部分。当编译 **Handle.cpp** 时，这个结构的定义被隐藏在目标文件中，没有人能看到它。如果改变了 **Cheshire** 的组成，惟一要重新编译的是 **Handle.cpp**，因为头文件并没有改动。

**Handle** 的使用就像任何类的使用一样，包含头文件、创建对象、发送消息。

```
//: C05:UseHandle.cpp
//{L} Handle
// Use the Handle class
#include "Handle.h"
```

```
int main() {
    Handle u;
    u.initialize();
    u.read();
    u.change(1);
    u.cleanup();
} ///:~
```

客户程序员惟一能访问的就是公共的接口部分，因此，如果只修改了在实现中的部分，上面文件就不须重新编译。虽然这并不是完全对实现进行了隐藏，但毕竟是一大改进。

## 5.7 小结

在C++中，访问控制为类的创建者提供了很有价值的控制。类的客户程序员可以清楚地看到，什么可以用，什么应该忽略。更重要的是，它保证了类的客户程序员不会依赖类的任何实现细节。有了这些，我们就可以更改类的实现部分，没有客户程序员会因此而受到影响，因为他们并不能访问类的这一部分。

一旦拥有了更改实现部分的自由，就可以在以后的时间里改进我们的设计，而且允许犯错误。要知道，无论如何小心地计划和设计，都可能犯错误。犯些错误也是相对安全的，这意味着我们会变得更有经验，会学得更快，就会更早完成项目。

一个类的公共接口部分是客户程序员能看到的。所以在分析设计阶段，保证接口的正确性更加重要。但这并不是说接口不能作修改。如果第一次没有正确地设计接口部分，可以再增加函数，这样就不需要删除那些已使用该类的程序代码。

## 5.8 练习

部分练习题的答案可以在本书的电子文档“*Annotated Solution Guide for Thinking in C++*”中找到，只需支付很少的费用就可以从<http://www.BruceEckel.com>得到这个电子文档。

5-1 创建一个类，具有 **public**、**private** 和 **protected** 数据成员和函数成员。创建该类的一个对象，看看当试图访问所有的类成员时会得到一些什么编译信息。

- 5-2 写一个名为**Lib**的**struct**，包括三个**string**对象**a**、**b**和**c**。在函数 **main()**中创建一个 **Lib**对象 **x**，对**x.a**、**x.b**、**x.c**赋值并打印出这些值。再用数组**string s[3]**代替 **a**、**b**、**c**。你将会看到由于这种改变，函数 **main()**中的代码出错。另外再创建一个名为**Libc**的类，有三个私有的 **string**对象**a**、**b**、**c**以及成员函数 **seta()**、**geta()**、**setb()**、**getb()**、**setc()**和**getc()**，这些成员函数用来设置和得到三个私有成员的值。像上面那样写一个函数 **main()**，现在把**private string**对象 **a**、**b**、**c**变成一个**private**数组**string s[3]**。这样你将会看到即使有这种改变，函数 **main()**中代码照样执行。
- 5-3 创建一个类和一个全局**friend**函数来处理类的**private**数据。
- 5-4 编写两个类，每个类都有一个成员函数，该函数中把一个指针指向另一个类的一个对象。在函数**main()**中创建两个实例对象，调用前面每一个类中的成员函数。
- 5-5 创建三个类。第一个类包括**private**数据，并且整个第二个类和第三个类的成员函数是它的友元，在函数**main()**中演示一下它们是如何正确运行的。
- 5-6 创建一个**Hen**类，在该类中，嵌套一个**Nest**类，在**Nest**类中，有一个**Egg**类成员。每一个类都有一个成员函数**display()**，在函数**main()**中创建每一个类的实例，然后调用每一个类的**display()**函数。
- 5-7 修改练习6中类**Nest**和类**Egg**，使它们都包含**private**数据，通过声明友元使套装类能够访问这些**private**数据。
- 5-8 创建一个有很多数据成员的类，这些数据成员分布在由**public**、**private**和**protected**所指定的区域中。增加一个成员函数**showMap()**，该成员函数打印这些数据成员的名字和它们的地址。如果有可能，在多个编译器、计算机或者操作系统中编译并运行这个程序，看目标代码中布局是否一样。
- 5-9 拷贝第4章中针对**Stash**的实现和测试文件，在本章中编译并测试**Stash.h**。
- 5-10 把来自练习6中**Hen**类的对象放到结构**Stash**中，取出并打印它们（如果还没有做，必须增加函数**Hen::print()**）。
- 5-11 拷贝第4章中针对**Stack**的实现和测试文件，在本章中编译并测试**Stack2.h**。
- 5-12 把来自练习6中**Hen**类的对象放到结构**Stack**中，取出并打印它们（如果还没有做，必须增加函数**Hen::print()**）。
- 5-13 修改**Handle.cpp**中的结构**Cheshire**，检验工程管理员是否只对这个文件进行了重新编译和重新连接，而不重新编译 **UseHandle.cpp**。
- 5-14 使用“Cheshire cat”技术创建类**StackOfInt**（一个存放整型数的堆栈），该技术隐藏了用于存储类**StackImp**中的元素的低级数据结构。实现**StackImp**有两种方法：一种是使用固定长的整型数组，另一种是使用**vector<int>**。在第一种方法中，由于通过预先调整设置了堆栈的最大尺寸，不必担心数组扩展。注意类**StackOfInt.h**不必随**StackImp**一起改变。



## 初始化与清除

第4章采用一个典型C语言库中所有分散的构件，并把它们封装进一个结构(一个抽象数据类型，从现在起，称其为一个类)，从而在库的应用方面作出了重大改进。

这样不仅为访问库构件提供了统一的入口，也用类名隐藏了类内部的函数名。在第5章中，我们介绍了访问控制(实现隐藏)。这就为类的设计者提供了一种设立边界的途径，通过边界的设立来决定哪些是允许客户程序员处理的，哪些是禁止客户程序员处理的。这意味着，对某种数据类型进行操作的内部机制处于类的设计者控制之下，可以由他们斟酌决定，这样也可以让客户程序员清楚哪些成员是他们能够使用并应该加以注意的。

封装和访问控制在改进库的易用性方面取得了重大进展。它们提供的“新的数据类型”的概念在某些方面比来自C语言的现有的内置数据类型要好。现在，C++编译器可以为这种新的数据类型提供类型检查保证，从而在使用这种数据类型时就确保了一定程度的安全性。

当然，说到安全性，C++的编译器能比C编译器提供更多的功能。在本章及以后的章节中，我们将看到C++的另外一些特征。它们可以让程序中的错误充分暴露，有时甚至在编译这个程序之前，帮助查出错误，但通常是编译器的警告和出错信息。基于这个原因，我们不久就会习惯于这样一种情景：一个C++程序在第一次编译时就能正确运行。

安全性问题包括初始化和清除两个方面。在C语言中，如果程序员忘记了初始化或清除一个变量，就会出现一大段程序错误。这在一个C库中尤其如此，特别是当客户程序员不知如何初始化一个**struct**，或甚至不知道他们必须要初始化一个**struct**时。(库中通常不包含初始化函数，所以客户程序员不得不自己手工初始化**struct**。)清除是一个特殊问题，因为C程序员一旦用过一个变量后就会把它忘记，所以对于一个库的**struct**来说必要的清除工作往往会被遗忘。

在C++中，初始化和清除的概念是简化库的使用的关键所在，并可以减少那些在客户程序员忘记去完成这些操作时会引起的细微错误。本章就来讨论C++的这些特征，它们有助于保证正常的初始化和清除。

### 6.1 用构造函数确保初始化

在**Stash**和**Stack**类中都曾调用**initialize()**函数，这个函数名暗示无论用什么方法使用这些对象都应当在对象使用之前调用这一函数。不幸的是，这要求客户程序员必须正确地初始化。而客户程序员在专注于用那令人惊奇的库来解决问题的时候，往往忽视了初始化的细节。在C++中，初始化实在太重要了，不应该留给客户程序员来完成。类的设计者可以通过提供一个叫做构造函数(*constructor*)的特殊函数来保证每个对象都被初始化。如果一个类有构造函数，编译器在创建对象时就自动调用这一函数，这一切在客户程序员使用他们的对象之前就已经完成了。是否调用构造函数不需要客户程序员来考虑，它是由编译器在对象定义时完成的。

接下来的问题是这个函数叫什么名字。这必须考虑两点，首先这个名字不能与类的其他成员函数冲突，其次，因为该函数是由编译器调用的，所以编译器必须总能知道调用哪个函

数。Stroustrup的方法似乎是最简单也最符合逻辑的：构造函数的名字与类的名字一样。这样的函数在初始化时会自动被调用。

下面是一个带构造函数的类的简单例子：

```
class X {
    int i;
public:
    X(); // Constructor
};
```

现在当一个对象被定义时：

```
void f() {
    X a;
    // ...
}
```

这时就好像是一个int一样：为这个对象分配内存。但是当程序执行到的序列点（*sequence point*）执行的点时，构造函数自动被调用，因为编译器已悄悄地在的定义点处插入了一个X::X()的调用。就像其他成员函数被调用一样。传递到构造函数的第一个（秘密）参数是this指针，也就是调用这一函数的对象的地址，不过，对构造函数来说，this指针指向一个没有被初始化的内存块，构造函数的作用就是正确的初始化该内存块。

像其他函数一样，也可以通过向构造函数传递参数，指定对象该如何创建或设定对象初始值，等等。构造函数的参数保证对象的所有部分都被初始化成合适的值。举例来说，如果类Tree有一个带整型参数的构造函数，用以指定树的高度，那么就必须这样来创建一个树对象：

```
Tree t(12); // 12-foot tree
```

如果Tree(int)是惟一的构造函数，编译器将不会用任何其他方法来创建一个对象（在下一章将看到多个构造函数以及调用它们的不同方法）。

关于构造函数就全部介绍完了。构造函数有着特殊的名字，在每个对象创建时，编译器自动调用的函数。尽管构造函数简单，但是它解决了类的很多问题，并使得代码更容易读写。例如在前面的代码段中，对有些initialize()函数并没有看到显式的调用，这些函数从概念上说是与定义分开的。在C++中，定义和初始化是集为一体的，不能只取其中之一。

构造函数和析构函数是两个非常特殊的函数：它们没有返回值。这与返回值为void的函数显然不同。后者虽然也不返回任何值，但还可以让它做点别的事情，而构造函数和析构函数则不允许。在程序中创建和消除一个对象的行为非常特殊，就像出生和死亡，而且总是由编译器来调用这些函数以确保它们被执行。如果它们有返回值，要么编译器必须知道如何处理返回值，要么就只能由客户程序员自己来显式地调用构造函数与析构函数，这样一来，安全性就被破坏了。

## 6.2 用析构函数确保清除

作为一个C程序员，可能经常想到初始化的重要性，但很少想到清除的重要性。毕竟，清除一个int时需要做什么？仅仅是忘记它。然而，在一个库中，对于一个曾经用过的对象，仅仅“忘记它”是不安全的。如果它修改了某些硬件参数，或在屏幕上显示了一些字符，或在堆中分配了一些内存，那么将会发生什么呢？如果只是“忘记它”，对象就永远不会消失。在

C++中，清除就像初始化一样重要。它通过析构函数来保证清除的执行。

析构函数的语法与构造函数一样，用类的名字作为函数名。然而析构函数前面加上一个代字号 (~)，以和构造函数区别。另外，析构函数不带任何参数，因为析构不需任何选项。下面是一个析构函数的声明：

```
class Y {
public:
    ~Y();
};
```

当对象超出它的作用域时，编译器将自动调用析构函数。可以看到，在对象的定义点处构造函数被调用，但析构函数调用的惟一证据是包含该对象的右括号。即使用goto语句跳出这一程序块（为了与C语言向后兼容，goto在C++中仍然存在，当然有时也是为了方便），析构函数仍然被调用。应该注意非局域的goto语句（nonlocal goto），它们是用标准C语言库中的setjmp()和longjmp()函数实现的，这些非局域的goto语句将不会引发析构函数的调用（这是一种规范：但有的编译器可能并不用这种方法来实现。对那些不在规范中的特征的依赖性意味着这样的代码是不可移植的）。

下例说明了构造函数与析构函数的上述特征：

```
//: C06:Constructor1.cpp
// Constructors & destructors
#include <iostream>
using namespace std;

class Tree {
    int height;
public:
    Tree(int initialHeight); // Constructor
    ~Tree(); // Destructor
    void grow(int years);
    void printsize();
};

Tree::Tree(int initialHeight) {
    height = initialHeight;
}

Tree::~~Tree() {
    cout << "inside Tree destructor" << endl;
    printsize();
}

void Tree::grow(int years) {
    height += years;
}

void Tree::printsize() {
    cout << "Tree height is " << height << endl;
}

int main() {
    cout << "before opening brace" << endl;
    {
```



```

    Tree t(12);
    cout << "after Tree creation" << endl;
    t.printsize();
    t.grow(4);
    cout << "before closing brace" << endl;
}
cout << "after closing brace" << endl;
} ///:~

```

下面是上面程序的输出结果：

```

before opening brace
after Tree creation
Tree height is 12
before closing brace
inside Tree destructor
Tree height is 16
after closing brace

```

可以看到析构函数在包括它的右括号处被调用。

### 6.3 清除定义块

在C中，总是要在一个程序块的左括号一开始就定义好所有的变量，这在程序设计语言中不算少见，其理由无非是因为“这是一种好的编程风格”。在这点上，我有自己的看法。我认为它总是带来不方便。作为一个程序员，每当需要增加一个变量时我都得跳到块的开始，我发现如果变量定义紧靠着变量的使用点时，程序的可读性更强。

也许这些争论仅限于格式。在C++中，是否一定要在块的开头就定义所有变量成了一个很突出的问题。如果存在构造函数，那么当对象产生时它必须首先被调用，如果构造函数带有一个或者更多个初始化参数，那么怎么知道在块的开头定义这些初始化信息呢？在一般的编程情况下将做不到这点，因为C中没有私有成员的概念。这样很容易将定义与初始化部分分开，然而C++要保证在一个对象产生时，它同时被初始化。这可以保证系统中没有未初始化的对象。C并不关心这些。事实上，C要求在块的开头定义变量，而这时还不知道一些必要的初始化信息<sup>①</sup>，这样就鼓励了不初始化变量的习惯。

通常，在C++中，在还不拥有构造函数的初始化信息时不能创建一个对象，所以不必在块的开头定义所有变量。事实上，这种语言风格似乎鼓励把对象的定义放得离使用点处尽可能近一点。在C++中，对一个对象适用的所有规则，对内建类型的对象也同样适用。这意味着任何类的对象或者内建类型的变量都可以在块的任何地方定义。这也意味着可以等到已经知道一个变量的必要信息时再去定义它，所以总是可以同时定义和初始化一个变量。

```

//: C06:DefineInitialize.cpp
// Defining variables anywhere
#include "../require.h"
#include <iostream>
#include <string>
using namespace std;

class G {

```

① 在标准C的升级版本C99中，可以像C++一样，在某一块的任意地方定义变量。

```

    int i;
public:
    G(int ii);
};
G::G(int ii) { i = ii; }

int main() {
    cout << "initialization value? ";
    int retval = 0;
    cin >> retval;
    require(retval != 0);
    int y = retval + 3;
    G g(y);
} ///:~

```

上例中可以看到先是执行一些代码，然后`retval`被定义和初始化，接着是一条用来接受客户程序员输入的语句，最后定义`y`和`g`。然而，在C中这些变量都只能在块的开始处定义。

一般说来，应该在尽可能靠近变量的使用点处定义变量，并在定义时就初始化（这是对内建类型的一种格式上的建议，而内建变量的初始化是可选的）。这是出于安全性的考虑，通过减少变量在块中的生命周期，就可以减少该变量在块的其他地方被误用的机会。另外，程序的可读性也增强了，因为读者不需要跳到块的开头去确定变量的类型。

### 6.3.1 for循环

在C++中，经常看到for循环的计数器直接在for表达式中定义：

```

for(int j = 0; j < 100; j++) {
    cout << "j = " << j << endl;
}
for(int i = 0; i < 100; i++)
    cout << "i = " << i << endl;

```

上述这些语句是一种重要的特殊情况，这可能使那些刚接触C++的程序员感到迷惑不解。

变量`i`和`j`都是在for表达式中直接定义的（在C中不能这样做），然后它们就可以作为一个变量在for循环中使用。这给程序员带来很大的方便，因为从上下文中我们可以清楚地知道变量`i`、`j`的作用，所以不必再用诸如`i_loop_counter`之类的名字来定义一个变量，以清晰地表示这一变量的作用。

然而，如果想把变量`i`、`j`的生命期扩展到for循环之外，就会有一些问题<sup>①</sup>。

在第3章中指出，`while`语句和`switch`语句也允许在它们的表达式内定义变量，尽管这种用法远没有for循环重要。

要注意局部变量会屏蔽其封闭块内的其他同名变量。通常，使用与全局变量同名的局部变量会使人产生误解，并且也易于产生错误<sup>②</sup>。

小作用域是良好设计的指标。如果一个函数有好几页，也许正在试图让这个函数完成太

① C++标准草案一个更早版本中，允许变量生命期扩展到包含for循环块的作用域。有些编译器仍旧这样实现，但它是不恰当的，所以我们的代码只有我们把块限制在for循环中时才可移植。

② Java语言认为这种做法不妥，将其认为是出错。

多的工作。如果用更多细化的函数，不仅更有用，而且更容易发现错误。

### 6.3.2 内存分配

现在，一个变量可以在某个程序范围内的任何地方定义，所以在这个变量的定义之前是无法对它分配内存空间的。通常，编译器更可能像C编译器一样，在一个程序块的开头就分配所有的内存。这些对我们来说是无关紧要的，因为作为一个程序员，在变量定义之前总是无法访问这块存储空间（即该对象）<sup>①</sup>。即使存储空间在块的一开始就被分配，构造函数也仍然要到对象的定义时才会被调用，因为标识符只有到此时才有效。编译器甚至会检查有没有把一个对象的定义（构造函数的调用）放到一个条件块中，比如在`switch`块中声明，或可能被`goto`跳过的地方。下例中解除注释的语句会导致一个警告或一个错误。

```
//: C06:Nojump.cpp
// Can't jump past constructors

class X {
public:
    X();
};

X::X() {}

void f(int i) {
    if(i < 10) {
        //! goto jump1; // Error: goto bypasses init
    }
    X x1; // Constructor called here
jump1:
    switch(i) {
        case 1 :
            X x2; // Constructor called here
            break;
        //! case 2 : // Error: case bypasses init
            X x3; // Constructor called here
            break;
    }
}

int main() {
    f(9);
    f(11);
}///:~
```

在上面的代码中，`goto`和`switch`都可能跳过构造函数调用的序列点，甚至构造函数没有被调用时，这个对象也会在后面的程序块中起作用，所以编译器给出了一条出错信息。这就确保了对象在产生的同时被初始化。

当然，这里讨论的内存分配都是在堆栈中进行的。内存分配是通过编译器向下移动堆栈指针来实现的（这里的“向下”是相对而言的，实际指针值增加，还是减少，取决于机器）。也可以在堆栈中使用`new`为对象分配内存，这将在第13章中进一步介绍。

① 当然，我们可以通过指针来访问这些存储空间，但这样做是非常有害的。

## 6.4 带有构造函数和析构函数的Stash

在前几章的例子中，都有一些很明显的函数对应为构造函数和析构函数：**initialize()**和**cleanup()**。下面是带有构造函数与析构函数的**Stash**头文件。

```
//: C06:Stash2.h
// With constructors & destructors
#ifndef STASH2_H
#define STASH2_H

class Stash {
    int size;           // Size of each space
    int quantity;       // Number of storage spaces
    int next;           // Next empty space
    // Dynamically allocated array of bytes:
    unsigned char* storage;
    void inflate(int increase);
public:
    Stash(int size);
    ~Stash();
    int add(void* element);
    void* fetch(int index);
    int count();
};
#endif // STASH2_H ///:~
```

下面是实现文件，这里只对**initialize()**和**cleanup()**的定义进行了修改，它们分别被构造函数与析构函数代替了。

```
//: C06:Stash2.cpp {0}
// Constructors & destructors
#include "Stash2.h"
#include "../require.h"
#include <iostream>
#include <cassert>
using namespace std;
const int increment = 100;

Stash::Stash(int sz) {
    size = sz;
    quantity = 0;
    storage = 0;
    next = 0;
}

int Stash::add(void* element) {
    if(next >= quantity) // Enough space left?
        inflate(increment);
    // Copy element into storage,
    // starting at next empty space:
    int startBytes = next * size;
    unsigned char* e = (unsigned char*)element;
    for(int i = 0; i < size; i++)
        storage[startBytes + i] = e[i];
    next++;
}
```



```

    return(next - 1); // Index number
}

void* Stash::fetch(int index) {
    require(0 <= index, "Stash::fetch (-)index");
    if(index >= next)
        return 0; // To indicate the end
    // Produce pointer to desired element:
    return &(storage[index * size]);
}

int Stash::count() {
    return next; // Number of elements in CStash
}

void Stash::inflate(int increase) {
    require(increase > 0,
        "Stash::inflate zero or negative increase");
    int newQuantity = quantity + increase;
    int newBytes = newQuantity * size;
    int oldBytes = quantity * size;
    unsigned char* b = new unsigned char[newBytes];
    for(int i = 0; i < oldBytes; i++)
        b[i] = storage[i]; // Copy old to new
    delete []storage; // Old storage
    storage = b; // Point to new memory
    quantity = newQuantity;
}

Stash::~Stash() {
    if(storage != 0) {
        cout << "freeing storage" << endl;
        delete []storage;
    }
} //::~~

```

**require.h**中的函数是用来监视程序员错误的，代替函数**assert()**的作用。但是函数**assert()**对失败操作的输出不及**require.h**的函数有效（关于这一点将在本书后面说明）。

因为**inflate()**是私有的，所以**require()**不能正确执行的惟一情况就是：其他成员函数意外地把一些不正确的值传递给了**inflate()**。如果能够确保这种情况不会发生，那么就考虑删除函数**require()**，但是在类稳定之前不要删除，当把新的代码加入到类中时，出错的可能性就会存在。由此看来，使用**require()**的代价很低（通过使用预处理器，有些代码能够被自动删除），这样代码也会具有很好的健壮性。

注意，在下面的测试程序中，**Stash**对象的定义放在紧靠使用对象的地方，对象的初始化通过构造函数的参数列表来实现，而对象的初始化似乎成了对象定义的一部分。

```

//: C06:Stash2Test.cpp
//{L} Stash2
// Constructors & destructors
#include "Stash2.h"
#include "../require.h"
#include <fstream>
#include <iostream>

```



```

#include <string>
using namespace std;

int main() {
    Stash intStash(sizeof(int));
    for(int i = 0; i < 100; i++)
        intStash.add(&i);
    for(int j = 0; j < intStash.count(); j++)
        cout << "intStash.fetch(" << j << ") = "
              << *(int*)intStash.fetch(j)
              << endl;
    const int bufsize = 80;
    Stash stringStash(sizeof(char) * bufsize);
    ifstream in("Stash2Test.cpp");
    assure(in, " Stash2Test.cpp");
    string line;
    while(getline(in, line))
        stringStash.add((char*)line.c_str());
    int k = 0;
    char* cp;
    while((cp = (char*)stringStash.fetch(k++))!=0)
        cout << "stringStash.fetch(" << k << ") = "
              << cp << endl;
} ///:~

```

再看看**cleanup()**调用已被取消，但当**intStash**和**stringStash**越出程序块的作用域时，析构函数被自动地调用了。

在**Stash**例子中需要注意的是：仅仅使用了内建类型，它们没有构造函数。如果试图将类对象拷贝到**Stash**中，就会出现很多问题，程序也不会正确执行。标准的C++库能够把对象正确地拷贝到使用它的容器中，但是，这是一个相当复杂的过程。在下面的**Stack**例子中，将会看到使用指针可以避免出现这种问题，在后面的章节中将修改**Stash**以便使用指针。

## 6.5 带有构造函数和析构函数的Stack

重新实现含有构造函数和析构函数的链表（在**Stack**内），看看使用**new**和**delete**时，构造函数和析构函数怎样巧妙地工作。这是修改后的头文件：

```

//: C06:Stack3.h
// With constructors/destructors
#ifndef STACK3_H
#define STACK3_H

class Stack {
    struct Link {
        void* data;
        Link* next;
        Link(void* dat, Link* nxt);
        ~Link();
    } * head;
public:
    Stack();
    ~Stack();
    void push(void* dat);
    void* peek();

```



```

    void* pop();
};
#endif // STACK3_H ///:~

```

不仅**Stack**有构造函数与析构函数，而且被嵌套的类**Link**也有。

```

//: C06:Stack3.cpp {0}
// Constructors/destructors
#include "Stack3.h"
#include "../require.h"
using namespace std;

Stack::Link::Link(void* dat, Link* nxt) {
    data = dat;
    next = nxt;
}

Stack::Link::~~Link() { }

Stack::Stack() { head = 0; }

void Stack::push(void* dat) {
    head = new Link(dat, head);
}

void* Stack::peek() {
    require(head != 0, "Stack empty");
    return head->data;
}

void* Stack::pop() {
    if(head == 0) return 0;
    void* result = head->data;
    Link* oldHead = head;
    head = head->next;
    delete oldHead;
    return result;
}

Stack::~~Stack() {
    require(head == 0, "Stack not empty");
} ///:~

```

构造函数**Link::Link()**只是简单地初始化**data**指针和 **next**指针，所以在**Stack::push()**中下面这行：

```
head = new Link(dat, head);
```

不仅为一个新的链表分配内存（用第4章中介绍的关键字**new**动态创建对象），而且也巧妙地初始化该对象的指针成员。

读者也许想知道**Link**的析构函数为什么不做任何事情，尤其是为什么不删除**data**指针？这里存在两个问题：在第4章引入**Stack**的地方，指出了如果**void**指针指向一个对象的话，就不能正确地将其删除（这种情况将在第13章中说明）。但是，除此之外，如果**Link**的析构函数删除**data**指针，**pop()**将最终返回一个指向被删除对象的指针，很明显，这会引入错误。有时这被看做是所有权（ownership）问题：**Link**和**Stack**仅仅存放指针，但它们不负责清除这些

指针。这意味着必须非常小心，要知道由谁来负责这种工作。例如：如果不做`pop()`而删除**Stack**中的所有指针，它们就不会自动被**Stack**的析构函数清除。这种问题不是独立的，它会导致内存泄漏，所以知道谁来负责清除一个对象，对于一个程序是成功还是失败来说是很关键的——这就是为什么如果**Stack**对象销毁时不为空，**Stack::~~Stack()**就会打印出错误信息的原因。

因为分配和清除**Link**对象的实现隐藏在类**Stack**中，它是内部实现的一部分，所以在测试程序中看不到它运行的结果，尽管从`pop()`返回的指针由我们负责删除。

```
//: C06:Stack3Test.cpp
//{L} Stack3
//{T} Stack3Test.cpp
// Constructors/destructors
#include "Stack3.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char* argv[]) {
    requireArgs(argc, 1); // File name is argument
    ifstream in(argv[1]);
    assure(in, argv[1]);
    Stack textlines;
    string line;
    // Read file and store lines in the stack:
    while(getline(in, line))
        textlines.push(new string(line));
    // Pop the lines from the stack and print them:
    string* s;
    while((s = (string*)textlines.pop()) != 0) {
        cout << *s << endl;
        delete s;
    }
} ///:~
```

既然这样，`textlines`中的所有行被弹出和删除，但是如果不出现这些操作的话，就会得到由`require()`带回的信息，这些信息表明这里有内存泄漏。

## 6.6 聚合初始化

顾名思义，聚合（*aggregate*）就是多个事物聚集在一起。这个定义包括混合类型的聚合：像**struct**和**class**等。数组就是单一类型的聚合。

初始化聚合往往既冗长又容易出错。而C++中聚合初始化（*aggregate initialization*）却变得很方便而且很安全。当产生一个聚合对象时，要做的只是指定初始值就行了，然后初始化工作就由编译器去承担了。这种指定可以用几种不同的风格，它取决于正在处理的聚合类型。但不管是哪种情况，指定的初值都要用大括号括起来。比如一个内建类型的数组可以这样定义：

```
int a[5] = { 1, 2, 3, 4, 5 };
```

如果给出的初始化值多于数组元素的个数，编译器就会给出一条出错信息。但如果给的初始化值少于数组元素的个数，那将会怎么样呢？例如：

```
int b[6] = {0};
```

这时，编译器会把第一个初始化值赋给数组的第一个元素，然后用0赋给其余的元素。注意，如果定义了一个数组而没有给出一列初始值时，编译器并不会去做初始化工作。所以上面的表达式是将一个数组初始化为零的简洁方法，它不需要用一个**for**循环，也避免了“偏移1位”错误（它可能比**for**循环更有效，这取决于编译器）。

数组还有一种叫自动计数（*automatic counting*）的快速初始化方法，就是让编译器按初始化值的个数去决定数组的大小：

```
int c[] = { 1, 2, 3, 4 };
```

现在，如果决定增加另一个元素到这个数组上，只要增加一个初始化值即可，如果以此建立我们的代码，只需在一处作出修改即可，这样，在修改时出错的机会就减少了。但怎样确定这个数组的大小呢？用表达式**sizeof c/sizeof \*c**（整个数组的大小除以第一个元素的大小）即可算出，这样，当数组大小改变时它不需要修改<sup>⊖</sup>。

```
for(int i = 0; i < sizeof c / sizeof *c; i++)
    c[i]++;
```

因为结构也是一种聚合类型，所以它们也可以用同样的方式初始化。因为C风格的**struct**的所有成员都是**public**型的，所以它们的值可以直接指定。

```
struct X {
    int i;
    float f;
    char c;
};
```

```
X x1 = { 1, 2.2, 'c' };
```

如果有一个这种**struct**的数组，也可以用嵌套的大括号来初始化每一个对象。

```
X x2[3] = { {1, 1.1, 'a'}, {2, 2.2, 'b'} };
```

这里，第三个对象被初始化为零。

如果**struct**中有私有成员（典型的情况就是C++中设计良好的类），或即使所有成员都是公共成员，但有构造函数，情况就不一样了。在上例中，初始值被直接赋给了聚合中的每个元素，但构造函数是通过正式的接口来强制初始化的。这里，构造函数必须被调用来完成初始化，因此，如果有一个下面的**struct**类型：

```
struct Y {
    float f;
    int i;
    Y(int a);
};
```

⊖ 在本书的第2卷中（可以在[http:// www.BruceEckel.com](http://www.BruceEckel.com)上免费获得），我们将会看到：通过使用模板可以更方便地决定一个数组的大小。

必须指示构造函数调用，最好的方法像下面这样：

```
Y y1[] = { Y(1), Y(2), Y(3) };
```

这样就得到了三个对象和进行了三次构造函数调用。只要有构造函数，无论是所有成员都是公共的**struct**还是一个带私有成员的**class**，所有的初始化工作都必须通过构造函数来完成，即使正在对一个聚合初始化。

下面是多构造函数参数的又一个例子：

```
//: C06:Multiarg.cpp
// Multiple constructor arguments
// with aggregate initialization
#include <iostream>
using namespace std;

class Z {
    int i, j;
public:
    Z(int ii, int jj);
    void print();
};

Z::Z(int ii, int jj) {
    i = ii;
    j = jj;
}

void Z::print() {
    cout << "i = " << i << ", j = " << j << endl;
}

int main() {
    Z zz[] = { Z(1,2), Z(3,4), Z(5,6), Z(7,8) };
    for(int i = 0; i < sizeof zz / sizeof *zz; i++)
        zz[i].print();
} ///:~
```

注意：这看起来就好像对数组中的每个对象都调用显式的构造函数。

## 6.7 默认构造函数

默认构造函数 (*default constructor*) 就是不带任何参数的构造函数。默认的构造函数用来创建一个“原型 (vanilla) 对象”，当编译器需要创建一个对象而又不知任何细节时，默认的构造函数就显得非常重要。比如，有一个前面定义的**struct Y**，并用它来定义对象：

```
Y y2[2] = { Y(1) };
```

编译器就会报告找不到默认的构造函数。数组中的第二个对象想不带参数来创建，在这里编译器就去找默认的构造函数。实际上，如果只是简单地定义了一个**Y**对象的数组：

```
Y y3[7];
```

这样编译的时候，就会出现错误，因为它必须有一个默认的构造函数来初始化数组中的每一个对象。

如果像下面一样单独创建一个对象时，也会出现同样的错误：

```
Y y4;
```

记住，一旦有了一个构造函数，编译器就会确保不管在什么情况下它总会被调用。

默认的构造函数非常重要，所以当（且仅当）在一个结构（**struct** 或 **class**）中没有构造函数时，编译器会自动为它创建一个。因此下面例子将会正常运行：

```
//: C06:AutoDefaultConstructor.cpp
// Automatically-generated default constructor

class V {
    int i; // private
}; // No constructor

int main() {
    V v, v2[10];
} ///:~
```

然而，一旦有构造函数而没有默认构造函数，上面的对象定义就会产生一个编译错误。

读者可能会想，由编译器合成的构造函数应该可以做一些智能化的初始化工作，比如把对象的所有内存置零。但事实并非如此。因为这样会增加额外的负担，而且使程序员无法控制。如果想把内存初始化为零，那就得显式地编写默认的构造函数。

尽管编译器会创建一个默认的构造函数，但是编译器合成的构造函数的行为很少是我们期望的。我们应该把这个特征看成是一个安全网，但尽量少用它。一般说来，应该明确地定义自己的构造函数，而不让编译器来完成。

## 6.8 小结

由C++提供的细致精巧机制应给我们这样一个强烈的暗示：在这个语言中，初始化和清除是多么至关重要的。在Stroustrup设计C++时，他所作的第一个有关C语言效率的观察就是，从很大程度上说，有关程序难题是由于没有适当地初始化变量而引起的。这种错误很难发现。同样的问题也出现在变量的清除上。因为构造函数与析构函数让我们保证正确地初始化和清除对象（编译器将不允许没有调用构造函数与析构函数就直接创建与销毁一个对象），使我们得到了完全的控制与安全。

聚合初始化同样如此——它防止犯那种初始化内建数据类型聚合时常犯的错误，使代码更简洁。

编码期间的安全性是C++中的一大问题，初始化和清除是这其中的一个重要部分，随着本书的深入学习，可以看到其他的安全性问题。

## 6.9 练习

部分练习题的答案可以在本书的电子文档“*Annotated Solution Guide for Thinking in C++*”中找到，只需支付很少的费用就可以从<http://www.BruceEckel.com>得到这个电子文档。

6-1 写一个简单的类**Simple**，其构造函数打印一些信息告诉我们它被调用。在函数**main()**中定义对象。

6-2 在练习1的类中增加一个析构函数，让它打印一些信息告诉我们它被调用。

- 6-3 修改练习2中的类，让它包含一个**int**成员。修改它的构造函数，让其带一个**int**参数，该参数的值存放在类的**int**成员中，构造函数和析构函数打印该整数的值。这样当对象创建和销毁时我们就可以看到。
- 6-4 写一个程序演示一下这种情况：当用**goto**跳出一个循环时，析构函数仍然被调用。
- 6-5 写两个**for**循环，用他们打印出0到10的值。对于第一个，在**for**循环之前定义循环计数器，而对于第二个，在**for**循环控制表达式中定义循环计数器。作为本练习的第二部分，修改第二个**for**循环的标识符使它与第一个循环的计数器的名字相同，编译程序，看看会得到什么结果。
- 6-6 修改第5章最后的文件**Handle.h**、**Handle.cpp**和**UseHandle.cpp**，以使用构造函数和析构函数。
- 6-7 使用聚合初始化创建一个**double**类型数组，指定其大小，但是并不提供所有的数组元素值，使用**sizeof**确定数组的大小并打印出这个数组，然后通过使用聚合初始化创建一个**double**类型数组并且自动地计算数组大小，然后打印这个数组。
- 6-8 使用聚合初始化创建一个**string**类对象数组，创建一个**Stack**用来存储这些字符串，逐步把数组中的元素压入**Stack**中，最后，从**Stack**中弹出并打印它们。
- 6-9 利用练习3中创建的对象数组演示自动计数和聚合初始化。在类中增加一个成员函数来打印一条信息。计算数组的大小，对数组的每个元素，调用新的成员函数。
- 6-10 创建一个没有构造函数的类，显示我们可以通过默认的构造函数创建对象。现在创建类的一个非默认的构造函数(带一个参数)，编译试试看。解释所发生的情况。



## 函数重载与默认参数

能使名字方便使用，是任何程序设计语言的一个重要特征。

当我们创建一个对象（一个变量）时，要为存储区取一个名字。函数就是一个操作的名字。通过编制各种名字来描述身边的系统，我们可以产生易于被人们理解和修改的程序。这在很大程度上就像是写文章——其目的是与读者进行交流。

这里就产生了这样一个问题：如何把人类自然语言中有细微差别的概念映射到程序设计语言中。通常，自然语言中同一个词可以代表多种不同的含义，具体含义要依赖上下文来确定。这就是所谓的一词多义——该词被重载（*overload*）了。这点非常有用，特别是对于细微的差别。我们可以说“洗衬衫，洗汽车”。如果非得说成“衬衫－洗衬衫，汽车－洗汽车”，那将是很愚蠢的，就好像听话的人对指定的动作毫无辨别能力一样。人类语言都有内在的冗余，所以即使漏掉几个词，我们仍然可以知道其中的含义。我们不需要惟一标识符——我们可以从上下文中理解它的含义。

然而，大多数程序设计语言要求我们为每个函数设定一个惟一标识符。如果我们想打印三种不同类型的数据：**int**、**char**和**float**，通常不得不创建三个不同的函数名，如**print\_int()**、**print\_char()**和**print\_float()**，这些既增加了我们的编程工作量，也给读者理解程序增加了困难。

在C++中，还有另外一个因素会使函数名重载：构造函数。因为构造函数的名字预先由类的名字确定，所以看上去只能有惟一一个构造函数名。但如果我们想用多种方法来创建一个对象时该怎么办呢？例如假设创建一个类，这个类可以用标准的方法初始化自身，也可以通过从文件中读取信息来初始化，我们需要两个构造函数，一个不带参数（默认构造函数），另一个以一个字符串作为参数，这个字符串是初始化对象的文件的名称。两个都是构造函数，所以它们必须有相同的名字：类名。因此，函数重载对于允许函数同名是必不可少的。在这种情况下，构造函数是与不同的参数类型一起使用的。

尽管函数重载对构造函数来说是必须的，但是它仍是一个通用的方便手段，并且可以与任意函数（不仅包括类成员函数）一起使用。另外，函数重载意味着，我们有两个库，它们都有同名的函数，只要它们的参数列表不同就不会发生冲突。我们将在本章中详细讨论所有这些问题。

本章的主题就是方便地使用函数名。函数重载允许多个函数同名，但还有第2种方法使函数调用更方便。如果我们想以不同的方法调用同一个函数，该怎么办呢？当函数有一个长长的参数列表时，而大多数参数每次调用都一样时，书写这样的函数调用会使人厌烦，程序可读性也差。C++中有一个很通用的特征叫做默认参数（*default argument*）。默认参数就是在用户调用一个函数时没有指定参数值而由编译器插入参数值的参数。因此，**f("hello")**、**f("hi",1)**和**f("howdy",2, 'c')**可以用来调用同一个函数。它们也可能用来调用三个已重载的函数，但当参数列表相同时，我们通常希望调用同一个函数来完成相同的操作。



函数重载和默认参数实际上并不复杂。当我们学习完本章时，我们就会明白什么时候要用到它们，以及编译、连接时它们是怎样实现的。

## 7.1 名字修饰

在第4章中介绍了名字修饰 (*name decoration*) 的概念。在下面的代码中：

```
void f();
class X { void f(); };
```

**class X**内的函数**f()**不会与全局的**f()**发生冲突，编译器用不同的内部名**f()**（全局函数）和**X::f()**（成员函数）来区分两个函数。在第4章中，我们建议在函数名前加类名的方法来命名函数，所以编译器使用的内部名字可能就是**\_f**和**\_X\_f**。函数名不仅与类名关系密切，而且还跟其他因素有关。

为什么要这样呢？假设重载了两个函数名：

```
void print(char);
void print(float);
```

无论这两个函数是某个类的成员函数，还是全局函数都无关紧要。如果编译器只使用函数名的域，编译器并不能产生惟一的内部标识符，这两种情况下都得用**\_print**结尾。重载函数的思想是让我们用同名的函数，但这些函数的参数列表应该不一样。所以，为了让重载函数正确工作，编译器要用不同的参数类型来修饰不同的函数名。上面的两个在全局范围定义的函数，可能会产生类似于**\_print\_char**和**\_print\_float**的内部名。因为，要注意编译器如何为这样的名字修饰没有统一的标准，所以不同的编译器可能会产生不同的内部名（让编译器产生汇编语言代码后就可以看到这个内部名是个什么样子了）。当然，如果想为特定的编译器和连接器购买编译过的库的话，这就会引起错误。但是即使名字修饰有统一的标准，因为编译器用不同的方式产生代码，也还会出现其他问题。

有关函数重载就讲到这里，可以对不同的函数用同样的名字，只要求函数的参数不同。编译器会修饰这些名字、范围和参数来产生内部名以供它和连接器使用。

### 7.1.1 用返回值重载

读了上面的介绍，我们自然会问：“为什么只能通过范围和参数来重载，为什么不能通过返回值呢？”乍一听，似乎完全可行，而且还用内部函数名修饰了返回值，然后就可以用返回值重载了：

```
void f();
int f();
```

当编译器能从上下文中惟一确定函数的意思时，如**int x = f();**这当然没有问题。然而，在C中，总是可以调用一个函数但忽略它的返回值，即调用了函数的副作用 (*side effect*)，在这种情况下，编译器如何知道调用哪个函数呢？更糟的是，读者怎么知道哪个函数会被调用呢？仅仅靠返回值来重载函数实在过于微妙了，所以在C++中禁止这样做。

### 7.1.2 类型安全连接

对名字修饰还可以带来一个额外的好处。在C中，如果用户错误地声明了一个函数，或者

更糟糕地，一个函数还没声明就调用了，而编译器则按函数被调用的方式去推断函数的声明。这是一个特别严重的问题。有时这种函数声明是正确的，但如果不正确，就会成为一个很难发现的错误。

在C++中，所有的函数在被使用前都必须事先声明，因此出现上述情况的机会大大减少了。编译器不会自动添加函数声明，所以我们应该包含一个合适的头文件。然而，假如由于某种原因还是错误地声明了一个函数，可能是通过自己手工声明，也可能是包含了一个错误的头文件（也许是一个过期的版本），名字修饰会给我们提供一个安全网，这也就是人们常说的类型安全连接（*type-safe linkage*）。

请看下面的几个例子。在第一个文件中，函数定义是：

```
//: C07:Def.cpp {0}
// Function definition
void f(int) {}
///:~
```

在第二个文件中，函数在错误的声明后调用：

```
//: C07:Use.cpp
//{L} Def
// Function misdeclaration
void f(char);

int main() {
    //! f(1); // Causes a linker error
} ///:~
```

即使知道函数实际上应该是**f(int)**，但编译器并不知道，因为它被告知——通过一个明确的声明——这个函数是**f(char)**。因此编译成功了，在C中，连接也能成功，但在C++中却不行。因为编译器会修饰这些名字，把它变成了诸如**f\_int**之类的名字，而使用的函数则是**f\_char**。当连接器试图找到**f\_char**引用时，它只能找到**f\_int**，所以它就会报告一条出错信息。这就是类型安全连接。虽然这种问题并不经常出现，但一旦出现就很难发现，尤其是在一个大项目中。这是利用C++编译器查找C语言程序中很隐蔽的错误的一个例子。

## 7.2 重载的例子

现在回过头来看看前面的例子，这里用重载函数来改写。如前所述，重载的一个很重要的应用是构造函数。可以在下面的**Stash**类中看到这一点。

```
//: C07:Stash3.h
// Function overloading
#ifndef STASH3_H
#define STASH3_H

class Stash {
    int size;           // Size of each space
    int quantity;       // Number of storage spaces
    int next;           // Next empty space
    // Dynamically allocated array of bytes:
    unsigned char* storage;
    void inflate(int increase);
public:
```

```

    Stash(int size); // Zero quantity
    Stash(int size, int initQuantity);
    ~Stash();
    int add(void* element);
    void* fetch(int index);
    int count();
};
#endif // STASH3_H ///:~

```

**Stash()**的第一个构造函数与前面一样，但第二个有一个**Quantity**参数指明分配内存位置的初始大小。在这个定义中，可以看到**quantity**的内部值与**storage**指针一起被置零。在第二个构造函数中，调用**inflate(initQuantity)**增大**quantity**的值可以指示被分配的存储空间的大小。

```

///: C07:Stash3.cpp {0}
// Function overloading
#include "Stash3.h"
#include "../require.h"
#include <iostream>
#include <cassert>
using namespace std;
const int increment = 100;

Stash::Stash(int sz) {
    size = sz;
    quantity = 0;
    next = 0;
    storage = 0;
}

Stash::Stash(int sz, int initQuantity) {
    size = sz;
    quantity = 0;
    next = 0;
    storage = 0;
    inflate(initQuantity);
}

Stash::~~Stash() {
    if(storage != 0) {
        cout << "freeing storage" << endl;
        delete []storage;
    }
}

int Stash::add(void* element) {
    if(next >= quantity) // Enough space left?
        inflate(increment);
    // Copy element into storage,
    // starting at next empty space:
    int startBytes = next * size;
    unsigned char* e = (unsigned char*)element;
    for(int i = 0; i < size; i++)
        storage[startBytes + i] = e[i];
    next++;
    return(next - 1); // Index number
}

```



```

void* Stash::fetch(int index) {
    require(0 <= index, "Stash::fetch (-)index");
    if(index >= next)
        return 0; // To indicate the end
    // Produce pointer to desired element:
    return &(storage[index * size]);
}

int Stash::count() {
    return next; // Number of elements in CStash
}

void Stash::inflate(int increase) {
    assert(increase >= 0);
    if(increase == 0) return;
    int newQuantity = quantity + increase;
    int newBytes = newQuantity * size;
    int oldBytes = quantity * size;
    unsigned char* b = new unsigned char[newBytes];
    for(int i = 0; i < oldBytes; i++)
        b[i] = storage[i]; // Copy old to new
    delete [] (storage); // Release old storage
    storage = b; // Point to new memory
    quantity = newQuantity; // Adjust the size
} ///:~

```

当用第一个构造函数时，没有内存分配给**storage**，内存是在第一次调用**add()**来增加一个对象时分配的，另外，当执行**add()**时，当前的内存块不够用时也会分配内存。

下面的测试程序中，两个构造函数都会被执行。

```

//: C07:Stash3Test.cpp
//{L} Stash3
// Function overloading
#include "Stash3.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
    Stash intStash(sizeof(int));
    for(int i = 0; i < 100; i++)
        intStash.add(&i);
    for(int j = 0; j < intStash.count(); j++)
        cout << "intStash.fetch(" << j << ") = "
              << *(int*)intStash.fetch(j)
              << endl;
    const int bufsize = 80;
    Stash stringStash(sizeof(char) * bufsize, 100);
    ifstream in("Stash3Test.cpp");
    assure(in, "Stash3Test.cpp");
    string line;
    while(getline(in, line))
        stringStash.add((char*)line.c_str());
}

```



```

    int k = 0;
    char* cp;
    while((cp = (char*)stringStash.fetch(k++))!=0)
        cout << "stringStash.fetch(" << k << ") = "
            << cp << endl;
} ///:~

```

对于**stringStash**调用构造函数，使用了第二个参数。假如知道需要解决的问题的一些情况，就可以为**Stash**选择初始大小。

### 7.3 联合

正如前面所看到的一样,在C++中, **struct**和**class**惟一的不同之处就在于, **struct**默认为**public**, 而**class**默认为**private**。很自然地, 也可以让**struct**有构造函数和析构函数。另外, 一个**union** (联合) 也可以带有构造函数、析构函数、成员函数甚至访问控制。在下面的例子中, 还能再一次看到使用重载的好处。

```

//: C07:UnionClass.cpp
// Unions with constructors and member functions
#include<iostream>
using namespace std;

union U {
private: // Access control too!
    int i;
    float f;
public:
    U(int a);
    U(float b);
    ~U();
    int read_int();
    float read_float();
};

U::U(int a) { i = a; }

U::U(float b) { f = b; }

U::~~U() { cout << "U::~~U()\n"; }

int U::read_int() { return i; }

float U::read_float() { return f; }

int main() {
    U X(12), Y(1.9F);
    cout << X.read_int() << endl;
    cout << Y.read_float() << endl;
} ///:~

```

从上面的代码中可以认为:**union**与**class**的惟一不同之处在于存储数据的方式(也就是说在**union**中**int**类型的数据和**float**类型的数据在同一内存区覆盖存放), 但是**union**不能在继承时作为基类使用, 从面向对象设计的观点来看, 这是一种极大的限制 (有关继承将在第14章中讨论)。

尽管成员函数使客户程序员对**union**的访问在一定程度上变得规范，但是，一旦**union**被初始化，仍然不能阻止他们选择错误的元素类型。例如在上面的程序中，即使不恰当，我们也可以写**X.read\_float()**，然而，一个更安全的**union**可以封装在一个类中。在下面的例子中，注意**enum**是如何阐明代码的，以及重载是如何同构造函数一起出现的。

```

//: C07:SuperVar.cpp
// A super-variable
#include <iostream>
using namespace std;

class SuperVar {
    enum {
        character,
        integer,
        floating_point
    } vartype; // Define one
    union { // Anonymous union
        char c;
        int i;
        float f;
    };
public:
    SuperVar(char ch);
    SuperVar(int ii);
    SuperVar(float ff);
    void print();
};

SuperVar::SuperVar(char ch) {
    vartype = character;
    c = ch;
}

SuperVar::SuperVar(int ii) {
    vartype = integer;
    i = ii;
}

SuperVar::SuperVar(float ff) {
    vartype = floating_point;
    f = ff;
}

void SuperVar::print() {
    switch (vartype) {
        case character:
            cout << "character: " << c << endl;
            break;
        case integer:
            cout << "integer: " << i << endl;
            break;
        case floating_point:
            cout << "float: " << f << endl;
            break;
    }
}

```



```
int main() {
    SuperVar A('c'), B(12), C(1.44F);
    A.print();
    B.print();
    C.print();
} ///:~
```

在上面的代码中，**enum**没有类型名（它是一个没有加标记的枚举），如果想立即定义**enum**的一个实例时，上面的这种做法是可取的。在这里以后没有必要涉及枚举的类型名，所以说，枚举的类型名是可选的，不是必须的。

**union**没有类型名和标识符。这叫做匿名联合（*anonymous union*），为这个**union**创建空间，但并不需要用标识符的方式和以点操作符（`.'`）方式访问这个**union**的元素。例如，如果匿名**union**是：

```
//: C07:AnonymousUnion.cpp
int main() {
    union {
        int i;
        float f;
    };
    // Access members without using qualifiers:
    i = 12;
    f = 1.22;
} ///:~
```

注意：我们访问一个匿名联合的成员就像访问普通的变量一样。惟一的区别在于：该联合的两个变量占用同一内存空间。如果匿名**union**在文件作用域内（在所有函数和类之外），则它必须被声明为**static**，以使它有内部的连接。

尽管**SuperVar**现在来说是安全的，但是，它的用途却有点值得怀疑，因为使用**union**的首要目的是为了节省空间，而增加**vartype**占用了**union**中很多与数据有关的空间。所以，节省的空间差不多就被抵消了。有两种选择可以使这种模式变得可行。如果**vartype**控制多个**union**实例——假如它们都是相同的数据类型——这样对于这一组实例，就仅仅只需要一个**vartype**，这样就不会占用更多的空间。一个更有效的方法是在所有**vartype**代码的前面加上**#ifdef**，这样就保证了在开发和测试中正确地使用。对于发行的代码，可以消除额外空间和时间开销。

## 7.4 默认参数

在**Stash3.h**中，比较了**Stash()**的两个构造函数，它们似乎并没有多大不同，对不对？事实上，第一个构造函数只不过是第二个的一个特例——它的初始**size**为零。在这种情况下创建和管理同一函数的两个不同版本实在是浪费精力。

C++通过默认参数（*default argument*）提供了一种补救方法。默认参数是在函数声明时就已给定的一个值，如果在调用函数时没有指定这一参数的值，编译器就会自动地插上这个值。在**Stash**的例子中，可以把两个函数：

```
Stash(int size); // Zero quantity
Stash(int size, int initQuantity);
```

用一个函数声明来代替：

```
Stash(int size, int initQuantity = 0);
```

这样，**Stash(int)**定义就简化掉了——所需要的是一个单一的**Stash(int,int)**定义。

现在这两个对象的定义：

```
Stash A(100), B(100, 0);
```

将会产生完全相同的结果。它们将调用同一个构造函数。但对于**A**，它的第二个参数是由编译器在看到第一个参数是**int**而且没有第二个参数时自动加上去的。编译器能看到默认参数，所以它知道应该允许这样的调用，就好像它提供第二个参数一样，而这第二个参数值就是已经告知编译器的默认参数。

默认参数同函数重载一样，给程序员提供了很多方便，它们都使我们可以不同的场合下使用同一函数名字。不同之处是，利用默认参数，当我们不想亲手提供这些值时，由编译器提供一个默认参数。上面的那个例子就是用默认参数而不用函数重载的一个很好的例子。否则，我们必然面临有几乎同样含义、同样操作的两个或更多的函数。当然，如果函数之间的行为差异较大，用默认参数就不合适了(对于这个问题，我们想知道两个差异较大的函数是否应当有相同的名字)。

在使用默认参数时必须记住两条规则。第一，只有参数列表的后部参数才是可默认的，也就是说，不可以在一个默认参数后面又跟一个非默认的参数。第二，一旦在一个函数调用中开始使用默认参数，那么这个参数后面的所有参数都必须是默认的（这可以从第一条中导出）。

默认参数只能放在函数声明中，通常在一个头文件中。编译器必须在使用该函数之前知道默认值。有时人们为了阅读方便在函数定义处放上一些默认的注释值。如：

```
void fn(int x /* = 0 */) { // ...
```

#### 7.4.1 占位符参数

函数声明时，参数可以没有标识符，当这些不带标识符的参数用做默认参数时，看起来很有意思。可以这样声明：

```
void f(int x, int = 0, float = 1.1);
```

在C++中，在函数定义时，并不一定需要标识符，如：

```
void f(int x, int, float flt) { /* ... */ }
```

在函数体中，**x**和**flt**可以被引用，但中间的这个参数值则不行，因为它没有名字。调用还必须为这个占位符（placeholder）提供一个值，有**f(1)**或**f(1,2,3.0)**。这种语法允许把一个参数用做占位符而不去用它。其目的在于以后可以修改函数定义而不需要修改所有的函数调用。当然，用一个有名字的参数也能达到同样的目的，但如果定义的这个参数在函数体内没有使用它，多数编译器会给出一条警告信息，并认为犯了一个逻辑错误。用这种没有名字的参数，我们就可以防止这种警告产生。

更重要的是，如果开始用了一个函数参数，而后来发现不需要用它，可以将它去掉而不会产生警告错误，而且不需要改动那些调用该函数以前版本的程序代码。



## 7.5 选择重载还是默认参数

函数重载和默认参数都给函数调用提供了方便。然而，有时它也会使人产生困惑：究竟该使用哪一种技术？例如，考虑下面的程序，它用来自动管理内存块。

```
//: C07:Mem.h
#ifndef MEM_H
#define MEM_H
typedef unsigned char byte;

class Mem {
    byte* mem;
    int size;
    void ensureMinSize(int minSize);
public:
    Mem();
    Mem(int sz);
    ~Mem();
    int msize();
    byte* pointer();
    byte* pointer(int minSize);
};
#endif // MEM_H ///:~
```

**Mem**对象包括一个**byte**块，以确保有足够的存储空间。默认的构造函数不分配任何的空间。第二个构造函数确保**Mem**对象中有**sz**大小的存储区，析构函数释放空间，**msize()**告诉我们当前**Mem**对象中还有多少字节，**pointer()**函数产生一个指向存储区起始地址的指针（**Mem**是一个相当底层的工具）。可以有一个重载版本的**pointer()**函数，用这个函数，客户程序员可以将一个指针指向一块内存。该块内存至少有**minSize**大，有成员函数能够做到这一点。

构造函数和**pointer()**成员函数都使用**private ensureMinSize()**成员函数来增加内存块的大小（请注意，如果内存块要调整的话，存放**pointer()**的结果是不安全的）。

下面是这个类的实现：

```
//: C07:Mem.cpp {0}
#include "Mem.h"
#include <cstring>
using namespace std;

Mem::Mem() { mem = 0; size = 0; }

Mem::Mem(int sz) {
    mem = 0;
    size = 0;
    ensureMinSize(sz);
}

Mem::~Mem() { delete []mem; }

int Mem::msize() { return size; }

void Mem::ensureMinSize(int minSize) {
    if(size < minSize) {
        byte* newmem = new byte[minSize];
```



```

        memset(newmem + size, 0, minSize - size);
        memcpy(newmem, mem, size);
        delete []mem;
        mem = newmem;
        size = minSize;
    }
}

byte* Mem::pointer() { return mem; }

byte* Mem::pointer(int minSize) {
    ensureMinSize(minSize);
    return mem;
} ///:~

```

可以看到，只有函数**ensureMinSize()**负责内存分配，它在第二个构造函数和函数**pointer()**的第二个重载形式中使用。如果**size**足够大的话，函数**ensureMinSize()**什么也不需要，为了使块变得大一些（可能会有这种情况，当使用默认构造函数时，块的大小为零），必须分配新的存储空间，使用标准的C语言库函数**memset()**把新分配的内存置零，关于这点已在第5章中作了介绍。接着调用标准C语言库函数**memcpy()**，在这种情况下，把已经存在于**mem**中内容拷贝到**newmem**中（通常用一种有效的方式），最后，删除旧的内存，然后把新的内存和大小赋给适当的成员。

设计**Mem**类的目的是把它作为其他类的一种工具，以简化它们的内存管理（例如，它可以隐藏由操作系统提供的更复杂的内存管理细节）。下面是一个测试程序，它创建了一个简单的“string”类：

```

//: C07:MemTest.cpp
// Testing the Mem class
//{L} Mem
#include "Mem.h"
#include <cstring>
#include <iostream>
using namespace std;

class MyString {
    Mem* buf;
public:
    MyString();
    MyString(char* str);
    ~MyString();
    void concat(char* str);
    void print(ostream& os);
};

MyString::MyString() { buf = 0; }

MyString::MyString(char* str) {
    buf = new Mem(strlen(str) + 1);
    strcpy((char*)buf->pointer(), str);
}

void MyString::concat(char* str) {
    if(!buf) buf = new Mem;

```



```

        strcat((char*)buf->pointer(
            buf->msize() + strlen(str) + 1), str);
    }

    void MyString::print(ostream& os) {
        if(!buf) return;
        os << buf->pointer() << endl;
    }

    MyString::~MyString() { delete buf; }

    int main() {
        MyString s("My test string");
        s.print(cout);
        s.concat(" some additional stuff");
        s.print(cout);
        MyString s2;
        s2.concat("Using default constructor");
        s2.print(cout);
    } ///:~

```

用这个类，所能做的是创建一个**MyString**，连接文本，打印输出到一个**ostream**中。该类仅仅包含了一个指向**Mem**的指针，但是请注意设置指针为零的默认构造函数和第二个构造函数的区别，第二个构造函数创建了一个**Mem**并把一些数据拷贝给它。使用默认构造函数的好处，就是可以非常便利地创建空值**MyString**对象的大数组，因为每一个对象只是一个指针，默认构造函数的惟一开销是赋零值。当连接数据时，**MyString**的开销才会开始增长。在此情况下，只有**Mem**对象不存在的情况下才会被创建。但是，要是使用默认的构造函数，并且从未连接任何数据，调用析构函数仍然是安全的，因为为零调用的**delete**已经定义，这样它不会试图释放存储空间，或者另外导致一些问题。

如果观察这两个构造函数，乍一看，好像这是默认构造函数最好的候选，然而，如果删除默认构造函数，像下面用一个默认的参数来写另外一个构造函数：

```
MyString(char* str = "");
```

将会发现，它能正常工作，但是，我们将会失去宝贵的效率，因为**Mem**对象总是会被创建。为了获得效率，必须修改构造函数：

```

MyString::MyString(char* str) {
    if(!*str) { // Pointing at an empty string
        buf = 0;
        return;
    }
    buf = new Mem(strlen(str) + 1);
    strcpy((char*)buf->pointer(), str);
}

```

这也意味着默认值变成了一个标志：使用非默认值将导致需执行的一块代码被单独分离。这样构造一个小的构造函数，虽然看起来很合理，但是一般会导致错误。如果必须查看默认值而不是把它当做一个普通值的话，这就会意味着实际上是在单个函数体中使用两个不同的有效的函数版本：一个版本用于正常情况，另一个版本用于默认情况。我们也许会把它当成两个完全不同的函数体，由编译器来选择究竟使用哪一个。这种做法会稍微提高程序的效率（但是通常情况下不易察觉），因为额外的参数不会被传递，特定条件下的代码也不会被执行。

更重要的是，我们使用两个完全不相干的函数维护两个函数的代码，而不是使用默认参数把它们组合成一个函数。这样，维护起来就更容易，尤其是当函数特别大时。

另外一方面，考虑一下**Mem**类，如果审视两个构造函数和两个**pointer()**函数时，可以发现：在两种情况下使用默认参数根本不会导致成员函数定义的改变。因此，类的定义可以如下面所示：

```
//: C07:Mem2.h
#ifndef MEM2_H
#define MEM2_H
typedef unsigned char byte;

class Mem {
    byte* mem;
    int size;
    void ensureMinSize(int minSize);
public:
    Mem(int sz = 0);
    ~Mem();
    int msize();
    byte* pointer(int minSize = 0);
};
#endif // MEM2_H ///:~
```

注意：调用**ensureMinSize(0)**总是非常有效。

尽管这两种情况都是基于效率问题作出决定的，但是应该注意不要陷入到只考虑效率的境地（这是诱人的）。设计类时，最重要的问题是类的接口（客户程序员可以使用的**public**成员）。如果产生的类容易使用和重用，那说明成功了。要是有必要，总是可以为了效率而作适当的调整。但是，如果程序员过分强调效率的话，设计的类的效果将是可怕的。应该主要关心的是接口清晰，使使用和阅读代码的人易于理解。注意**MemTest.cpp**文件中**MyString**的语法没有变化，不管一个默认的构造函数是否使用，以及效率是高还是低。

## 7.6 小结

不能把默认参数作为一个标志去决定执行函数的哪一块，这是基本原则。在这种情况下，只要能够，就应该把函数分解成两个或多个重载的函数。一个默认的参数应该是一个在一般情况下放在这个位置的值。这个值出现的可能比其他值要大，所以客户程序员可以忽略它或只在需要改变默认值时才去用它。

默认参数的引用是为了使函数调用更容易，特别是当这些函数的许多参数都有特定值时。它不仅使书写函数调用更容易，而且阅读也更方便，尤其是当类的创建者能够制定参数，以便把那些最不可能调整的默认参数放在参数表的最后面时。

默认参数的一个重要应用情况是在开始定义函数时用了一组参数，而使用了一段时间后发现要增加一些参数。通过把这些新增参数都作为默认的参数，就可以保证所有使用这一函数的客户代码不会受到影响。

## 7.7 练习

部分练习题的答案可以在本书的电子文档“*Annotated Solution Guide for Thinking in C++*”

中找到，只需支付很小的费用就可以从<http://www.BruceEckel.com>得到这个电子文档。

- 7-1 创建一个包含一个**string**对象的**Text**类，来保存一个文件的内容。写两个构造函数：一个是默认的构造函数，另一个构造函数带有一个**string**参数，它是要打开的文件的名字。当使用第二个构造函数时，打开这个文件并把内容读到**string**成员对象中。增加一个成员函数**contents()**用来返回**string**，以便可以打印。在**main()**函数中，使用**Text**打开一个文件并打印该文件的内容。
- 7-2 创建一个**Message**类，其构造函数带有一个**string**型的默认参数。创建一个私有成员**string**，在构造函数中只是简单地把参数**string**赋值给内部的**string**。创建两个重载的成员函数**print()**：一个不带参数，而只是显示存储在对象中的信息；另一个带有**string**型参数，它将显示该字符串加上对象内部信息。比较这种方法和使用构造函数的方法，看哪种方法更合理？
- 7-3 确定您的编译器是怎样产生汇编输出代码的，并运行实验以观察名字修饰表。
- 7-4 创建带有4个成员函数的类，4个成员函数分别带有0、1、2、3个**int**参数。创建**main()**函数，产生你的类对象并调用每一个成员函数。然后修改类，使它只有一个成员函数，并且都使用默认参数。你的**main()**函数需要改变吗？
- 7-5 创建带有两个参数的函数，在**main()**中调用它。然后让一个参数作为“占位符”（没有标识符），看看**main()**中的调用是否改变。
- 7-6 用默认参数修改**Stash3.h**和**Stash3.cpp**中的构造函数，创建两个不同的**Stash**对象来测试构造函数。
- 7-7 创建一个新的**Stack**类（见第6章），默认构造函数如前面所述，还有第二个构造函数，它的参数是指向对象的指针数组和数组的大小。该构造函数应该遍历数组并把指针压入**Stack**中，用一个**string**数组测试你的程序。
- 7-8 修改**SuperVar**以便在所有**vartype**代码前有**#ifdef**，描述见前面关于**enum**的章节。让**vartype**成为一个常规的**public**枚举类型（没有实例），修改**print()**，使得它要求**vartype**参数能告诉它做什么。
- 7-9 实现**Mem2.h**，确保修改的类仍旧能与**MemTest.cpp**一起工作。
- 7-10 使用**Mem**类来实现**Stash**。注意：由于该实现是**private**，因此用户看不到，测试代码不必修改。
- 7-11 在**Mem**类中，增加一个**bool**类型的成员函数**moved()**。它引用**pointer()**的结果，告诉指针是否已经移动（由于被重新分配）。写一个**main()**函数来测试**moved()**函数。每次需要访问**Mem**中的内存时，是使用像**moved()**这样的函数好，还是简单地调用**pointer()**好？

## 常 量

常量概念（由关键字**const**表示）是为了使程序员能够在变和不变之间画一条界线。这在C++程序设计项目中提供了安全性和可控性。

自从常量概念出现以来，它就有多种不同的用途。与此同时，常量的概念慢慢地渗回到C语言中（在C语言中，它的含义已经改变）。在开始时，所有这些看起来是有点混淆。在本章里，将介绍什么时候、为什么和怎样使用关键字**const**。最后讨论关键字**volatile**，它是**const**的“近亲”（因为它们都关系到变化）并具有完全相同的语法。

**const**的最初动机是取代预处理器**#defines**来进行值替代。从这以后它曾被用于指针、函数变量、返回类型、类对象以及成员函数。所有这些用法都稍有区别，但它们在概念上是一致的，我们将在以下各节中说明这些用法。

## 8.1 值替代

当用C语言进行程序设计时，预处理器可以不受限制地建立宏并用它来替代值。因为预处理器只做些文本替代，它既没有类型检查概念，也没有类型检查功能，所以预处理器的值替代会产生一些微小的问题，这些问题在C++中可以通过使用**const**值而避免。

预处理器在C语言中用值替代名字的典型用法是这样的：

```
#define BUFSIZE 100
```

**BUFSIZE**是一个名字，它只是在预处理期间存在，因此它不占用存储空间且能放在一个头文件里，目的是为使用它的所有编译单元提供一个值。使用值替代而不是使用所谓的“不可思议的数”，这对于支持代码维护是非常重要的。如果代码中用到不可思议的数，读者不仅不清楚这个数字来自哪里，而且也不知道它代表什么。进而，当决定改变一个值时，程序员必须进行手工编辑，而且还不能跟踪以保证没有漏掉其中的一个（或者不小心改变了一个不应该改变的值）。

大多数情况，**BUFSIZE**的工作方式与普通变量类似；而且没有类型信息。这就会隐藏一些很难发现的错误。C++用**const**来消除这些问题，具体方法是把值替代移交给编译器。那么可以这样写：

```
const int bufsize = 100;
```

这样就可以在编译时编译器需要知道这个值的任何地方使用**bufsize**，同时编译器还可以执行常量折叠（*constant folding*），也就是说，编译器在编译时可以通过必要的计算把一个复杂的常量表达式通过缩减简单化。这一点在数组定义里显得尤其重要：

```
char buf[bufsize];
```

可以为所有的内建数据类型（**char**、**int**、**float**和**double**型）以及由它们所定义的变量（也可以是类的对象，这将在以后章节里讲到）使用限定符**const**。因为预处理器会引入错误，所以我们应该完全用**const**取代**#define**的值替代。

### 8.1.1 头文件里的const

要使用**const**而非**#define**，同样必须把**const**定义放进头文件里。这样，通过包含头文件，可把**const**定义单独放在一个地方并把它分配给一个编译单元。C++中的**const**默认为内部连接 (*internal linkage*)，也就是说，**const**仅在**const**被定义过的文件里才是可见的，而在连接时不能被其他编译单元看到。当定义一个**const**时，必须赋一个值给它，除非用**extern**作出了清楚的说明：

```
extern const int bufsize;
```

通常C++编译器并不为**const**创建存储空间，相反它把这个定义保存在它的符号表里。但是，上面的**extern**强制进行了存储空间分配（另外还有一些情况，如取一个**const**的地址，也要进行存储空间分配），由于**extern**意味着使用外部连接，因此必须分配存储空间，这也就是说有几个不同的编译单元应当能够引用它，所以它必须有存储空间。

通常情况下，当**extern**不是定义的一部分时，不会分配存储空间。如果使用**const**，那么编译时会进行常量折叠。

当然，想绝对不为任何**const**分配存储是不可能的，尤其对于复杂的结构。在这种情况下，编译器建立存储，这会阻止常量折叠（因为没有办法让编译器确切地知道内存的值是什么——要是知道的话，它也不必分配内存了）。

由于编译器不能完全避免为**const**分配内存，所以**const**的定义必须默认内部连接，即连接仅在特定的编译单元内；否则，由于众多的**const**在多个**cpp**文件内分配存储，容易引起连接错误，连接程序在多个对象文件里看到同样的定义就会“抱怨”。然而，因为**const**默认内部连接，所以连接程序不会跨过编译单元连接那些定义，因此不会有冲突。在大部分场合使用内建数据类型的情况，包括常量表达式，编译都能执行常量折叠。

### 8.1.2 const的安全性

**const**的作用不仅限于在常数表达式里代替**#defines**。如果用运行期间产生的值初始化一个变量而且知道在变量生命期内是不变的，则用**const**限定该变量是程序设计中的一个很好的做法。如果偶然试图改变它，编译器会给出出错信息。下面是一个例子：

```
//: C08:Safecons.cpp
// Using const for safety
#include <iostream>
using namespace std;

const int i = 100; // Typical constant
const int j = i + 10; // Value from const expr
long address = (long)&j; // Forces storage
char buf[j + 10]; // Still a const expression

int main() {
    cout << "type a character & CR:";
    const char c = cin.get(); // Can't change
    const char c2 = c + 'a';
    cout << c2;
    // ...
} ///:~
```



我们会发现，**i**是一个编译期间的**const**，但**j**是从**i**中计算出来的。然而，由于**i**是一个**const**，**j**的计算值来自一个常数表达式，而它自身也是一个编译期间的**const**。紧接下面的一行需要**j**的地址，所以迫使编译器给**j**分配存储空间。即使分配了存储空间，把**j**值保存在程序的某个地方，由于编译器知道**j**是**const**，而且知道**j**值是有效的，因此，这仍不能妨碍在决定数组**buf**的大小时使用**j**。

在主函数**main()**里，对于标识符**c**有另一种**const**，因为其值在编译期间是不知道的。这意味着需要存储空间，而编译器不想保留它的符号表里的任何东西（和C语言的行为一样）。初始化必须在定义点进行，而且一旦初始化，其值就不能改变。我们看到**c2**由**c**的值计算出来，也会看到这类常量的作用域与其他任何类型**const**的作用域是一样的——这是对**#define**用法的另一种改进。

就实际来说，如果想让一个值不变，就应该使之成为**const**。这不仅为防止意外的更改提供安全措施，也消除了读存储器和读内存操作，使编译器产生的代码更有效。

### 8.1.3 聚合

**const**可以用于聚合，你要相信编译器不会真的把一个聚合保存到它的符号表中，所以必须分配内存。在这种情况下，**const**意味着“不能改变的一块存储空间”。然而，不能在编译期间使用它的值，因为编译器在编译期间不需要知道存储的内容。这样，就能明白下面的代码是非法的：

```
//: C08:Constag.cpp
// Constants and aggregates
const int i[] = { 1, 2, 3, 4 };
//! float f[i[3]]; // Illegal
struct S { int i, j; };
const S s[] = { { 1, 2 }, { 3, 4 } };
//! double d[s[1].j]; // Illegal
int main() {} ///:~
```

在一个数组定义里，编译器必须能产生这样的代码，它们移动栈指针来存储数组。在上面这两种非法定义里，编译器给出“提示”是因为它不能在数组定义里找到一个常数表达式。

### 8.1.4 与C语言的区别

常量引进是在C++的早期版本中，当时标准C规范正在制定。那时，尽管C委员会决定在C中引入**const**，但是，不知何故，对他们来说，C中**const**的意思是“一个不能被改变的普通变量”，**const**常量总是占用存储而且它的名字是全局符。这样，C编译器不能把**const**看成一个编译期间的常量。在C中，如果写：

```
const int bufsize = 100;
char buf[bufsize];
```

尽管看起来好像做了一件合理的事，但这将得出一个错误。因为**bufsize**占用某块内存，所以C编译器不知道它在编译时的值。在C语言中可以选择这样书写：

```
const int bufsize;
```



这样写在C++中是不对的，而C编译器则把它作为一个声明，指明在别的地方有存储分配。因为C默认**const**是外部连接的，所以这样做是合理的。C++默认**const**是内部连接的，这样，如果在C++中想完成与C中同样的事情，必须用**extern**明确地把连接改成外部连接：

```
extern const int bufsize; // Declaration only
```

这行代码也可用在C语言中。

在C++中，一个**const**不必创建内存空间，而在C中，一个**const**总是需要创建一块内存空间。在C++中，是否为**const**常量创建内存空间依赖于对它如何使用。一般说来，如果一个**const**仅仅用来把一个名字用一个值代替（如同使用**#define**一样），那么该存储空间就不必创建。要是存储空间没有创建的话（这依赖于数据类型的复杂性以及编译器的性能），在进行完数据类型检查之后，为了代码更加有效，值也许会折叠到代码中，这和以前使用**#define**不同。不过，如果取一个**const**的地址（甚至不知不觉地把它传递给一个带引用参数的函数）或者把它定义成**extern**，则会为该**const**创建内存空间。

在C++中，出现在所有函数之外的**const**的作用域是整个文件（也就是它只是在该文件外不可见），也就是说，它默认为内部连接，这和C++中的所有其他默认为外部连接标识符很不一样（也与C中的**const**不一样）。因此，如果在两个不同文件中声明同名的**const**不取它的地址，也不把它定义成**extern**，那么理想的C++编译器就不会为它分配内存空间，而只是简单地把它折叠到代码中。因为**const**在一个文件范围内有效，所以可以把它放在C++头文件中，在连接时不会造成任何冲突。

因为C++中的**const**默认为内部连接，所以不能在一个文件中定义一个**const**，而在另外一个文件中又把它作为**extern**来引用。为了使**const**成为外部连接以便让另外一个文件可以对它引用，必须明确地把它定义成**extern**，如下面这样：

```
extern const int x = 1;
```

注意，通过对它进行初始化并指定为**extern**，我们强迫给它分配内存（虽然编译器在这里仍然可以选择常量折叠）。初始化使它成为一个定义而不是一个声明。在C++中的声明：

```
extern const int x;
```

意味着在别处进行了定义（在C中，不一定这样）。现在明白为什么C++要求一个**const**定义时需要初始化：初始化把定义和声明区别开来（在C中，它总是一个定义，所以初始化不是必需的）。当进行了**extern const**声明时，编译器就不能够进行常量折叠了，因为它不知道具体的值。

在C语言中使用限定符**const**不是很有用的，如果希望在常数表达式里（必须在编译期间被求值）使用一个已命名的值，C总是迫使程序员在预处理器里使用**#define**。

## 8.2 指针

还可以使指针成为**const**。当处理**const**指针时，编译器仍将努力避免存储分配并进行常量折叠，但在这种情况下，这些特征似乎很少有用。更重要的是，如果程序员以后想在程序代码中改变**const**这种指针的使用，编译器将给出通知。这大大增加了安全性。

当使用带有指针的**const**时，有两种选择：**const**修饰指针正指向的对象，或者**const**修饰在指针里存储的地址。这些语法在开始时有点使人混淆，但实践之后就好了。

### 8.2.1 指向const的指针

正如任何复杂的定义一样，定义指针的技巧是在标识符的开始处读它并从里向外读。**const**修饰“最靠近”它的那个。这样，如果要使正指向的元素不发生改变，得写一个像这样的定义：

```
const int* u;
```

从标识符开始，是这样读的：“**u**是一个指针，它指向一个**const int**。”这里不需要初始化，因为**u**可以指向任何标识符（也就是说，它不是一个**const**），但它所指的值是不能被改变的。

这是一个容易混淆的部分。有人可能认为：要想指针本身不变，即包含在指针**u**里的地址不变，可简单地像这样把**const**从**int**的一边移向另一边：

```
int const* v;
```

并非所有的人都很肯定地认为：应该读成“**v**是一个指向**int**的**const**指针”。然而，实际上应读成“**v**是一个指向恰好是**const**的**int**的普通指针”。即**const**又把它自己与**int**结合在一起，效果与前面定义一样。两个定义是一样的，这一点容易使人混淆。为使程序更具有可读性，应该坚持用第一种形式。

### 8.2.2 const指针

使指针本身成为一个**const**指针，必须把**const**标明的部分放在\*的右边，如：

```
int d = 1;
int* const w = &d;
```

现在它读成“**w**是一个指针，这个指针是指向**int**的**const**指针”。因为指针本身现在是**const**指针，编译器要求给它一个初始值，这个值在指针生命期间内不变。然而要改变它所指向的值是可以的，可以写

```
*w = 2;
```

也可以使用下面两种合法形式中的任何一种把一个**const**指针指向一个**const**对象：

```
int d = 1;
const int* const x = &d; // (1)
int const* const x2 = &d; // (2)
```

现在，指针和对象都不能改变。

一些人认为第二种形式的一致性更好，因为**const**总是放在被修饰者的右边。但对于特定的编码风格来讲，程序员应当自己决定哪一种形式更清楚。

下面这个可编译的文件包含上面出现的一些语句

```
//: C08:ConstPointers.cpp
const int* u;
int const* v;
int d = 1;
int* const w = &d;
const int* const x = &d; // (1)
int const* const x2 = &d; // (2)
int main() {} ///:~
```

### 8.2.2.1 格式

本书主张：只要可能，一行只定义一个指针，并尽可能在定义时初始化。正因为这一点，才可以把‘\*’“附于”数据类型上：

```
int* u = &i;
```

**int\***本身好像是一个离散类型。这使代码更容易懂，可惜的是，实际上事情并非那样。事实上，‘\*’与标识符结合，而不是与类型结合。它可以被放在类型名和标识符之间的任何地方。所以，可以这样做：

```
int *u = &i, v = 0;
```

它建立一个**int\* u**和一个非指针**int v**。由于读者时常混淆这一点，因此最好用本书里所用的表示形式（即一行里只定义一个指针）。

### 8.2.3 赋值和类型检查

C++关于类型检查是非常精细的，这一点也扩展到指针赋值。可以把一个非**const**对象的地址赋给一个**const**指针，因为也许有时不想改变某些可以改变的东西。然而，不能把一个**const**对象的地址赋给一个非**const**指针，因为这样做可能通过被赋值的指针改变这个对象的值。当然，总能用类型转换强制进行这样的赋值，但是，这是一个不好的程序设计习惯，因为这样就打破了对对象的**const**属性以及由**const**提供的安全性。例如：

```
//: C08:PointerAssignment.cpp
int d = 1;
const int e = 2;
int* u = &d; // OK -- d not const
//! int* v = &e; // Illegal -- e const
int* w = (int*)&e; // Legal but bad practice
int main() {} ///:~
```

虽然C++有助于防止错误发生，但如果程序员自己打破了这种安全机制，它也是无能为力的。

#### 8.2.3.1 字符数组的字面值

没有强调严格的**const**特性的地方，是字符数组的字面值。也许有人可以写：

```
char* cp = "howdy";
```

编译器将接受它而不报告错误。从技术上讲，这是一个错误，因为字符数组的字面值（这里是“**howdy**”）是被编译器作为一个常量字符数组建立的，所引用该字符数组得到的结果是它在内存里的首地址。修改该字符数组的任何字符都会导致运行时错误，当然，并不是所有的编译器都会做到这一点。

所以字符数组的字面值实际上是常量字符数组。当然，编译器把它们作为非常量看待，这是因为有许多现有的C代码是这样做的。当然，改变字符数组的字面值的做法还未被定义，虽然可能在很多机器上是这样做的。

如果想修改字符串，就要把它放到一个数组中：

```
char cp[] = "howdy";
```

因为编译器常常不强调它们的差别，所以可以不使用后面这种形式，在这一点上已变得无关紧要了。

### 8.3 函数参数和返回值

用**const**限定函数参数及返回值是常量概念容易引起混淆的另一个地方。如果按值传递对象，对客户来讲，用**const**限定没有意义（它意味着传递的参数在函数里是不能被修改的）。如果按常量返回用户定义类型的一个对象的值，这意味着返回值不能被修改。如果传递并返回地址，**const**将保证该地址内容不会被改变。

#### 8.3.1 传递**const**值

如果函数参数是按值传递，则可用指定参数是**const**的，如：

```
void f1(const int i) {
    i++; // Illegal -- compile-time error
}
```

这是什么意思呢？这是作了一个约定：变量初值不会被函数**f1()**改变。然而，由于参数是按值传递的，因此要立即产生原变量的副本，这个约定对客户来说是隐式的。

在函数里，**const**有这样的意义：参数不能被改变。所以它其实是函数创建者的工具，而不是函数调用者的工具。

为了不使调用者混淆，在函数内部用**const**限定参数优于在参数表里用**const**限定参数。可以用一个指针来实现，但更好的语法形式是“引用”，这是第11章讨论的主题。简而言之，引用像一个被自动间接引用的常量指针，它的作用是成为对象的别名。为建立一个引用，在定义里使用**&**。所以，不引起混淆的函数定义应该是这样的：

```
void f2(int ic) {
    const int& i = ic;
    i++; // Illegal -- compile-time error
}
```

这又会得到一个错误信息，但这时局部对象的常量性（**constness**）不是函数特征标志的部分；它仅对函数实现有意义，所以它对客户来说是不可见的。

#### 8.3.2 返回**const**值

对返回值来讲，存在一个类似的道理，即如果一个函数的返回值是一个常量（**const**）：

```
const int g();
```

这就约定了函数框架里的原变量不会被修改。另外。因为这是按值返回的，所以这个变量被制成副本，使得初值不会被返回值所修改。

首先，这使**const**看起来没有什么意义。可以从这个例子中看到：按值返回**const**明显失去作用：

```
//: C08:Constval.cpp
// Returning consts by value
// has no meaning for built-in types
```

```

int f3() { return 1; }
const int f4() { return 1; }

int main() {
    const int j = f3(); // Works fine
    int k = f4(); // But this works fine too!
} ///:~

```

对于内建类型来说，按值返回的是不是一个**const**，是无关紧要的，所以按值返回一个内建类型时，应该去掉**const**，从而不使客户程序员混淆。

当处理用户定义的类型时，按值返回常量是很重要的。如果一个函数按值返回一个类对象为**const**时，那么这个函数的返回值不能是一个左值（即它不能被赋值，也不能被修改）。例如：

```

//: C08:ConstReturnValues.cpp
// Constant return by value
// Result cannot be used as an lvalue

class X {
    int i;
public:
    X(int ii = 0);
    void modify();
};

X::X(int ii) { i = ii; }

void X::modify() { i++; }

X f5() {
    return X();
}

const X f6() {
    return X();
}

void f7(X& x) { // Pass by non-const reference
    x.modify();
}

int main() {
    f5() = X(1); // OK -- non-const return value
    f5().modify(); // OK
    ///! f7(f5()); // Causes warning
    // Causes compile-time errors:
    ///! f6() = X(1);
    ///! f6().modify();
    ///! f7(f6());
} ///:~

```

**f5()** 返回一个非**const X**对象，然而**f6()** 返回一个**const X**对象。仅仅是非**const**返回值能作为一个左值使用，因此，当按值返回一个对象时，如果不让这个对象作为一个左值使用，则使用**const**很重要。

当按值返回一个内建类型时，**const**没有意义的原因是：编译器已经不让它成为一个左值

(因为它总是一个值而不是一个变量)。仅当按值返回用户定义的类型对象时,才会出现上述问题。

函数f7()把它的参数作为一个非const引用(reference)(C++中另一种处理地址的办法,这是第11章讨论的主题)。从效果上讲,这与取一个非const指针一样,只是语法不同。在C++中不能编译通过的原因是会产生一个临时量。

#### 8.3.2.1 临时量

有时候,在求表达式值期间,编译器必须创建临时对象(temporary object)。像其他任何对象一样,它们需要存储空间,并且必须能够构造和销毁。区别是从来看不到它们——编译器负责决定它们的去留以及它们存在的细节。但是关于临时量有这样一种情况:它们自动地成为常量。通常接触不到临时对象,改变临时量是错误的,因为这些信息应该是不可得的。编译器使所有的临时量自动地成为const,这样当程序员犯那样的错误时,会向他发出错误警告。

在上面的例子中,f5()返回一个非const X对象,但是在表达式:

```
f7(f5());
```

中,编译器必须产生一个临时对象来保存f5()的返回值,使得它能传递给f7()。如果f7()的参数是按值传递的话,它能很好地工作,然后在f7()中形成那个临时量的副本,不会对临时对象X产生任何影响。但是,如果f7()的参数是按引用传递的,这意味着它取临时对象X的地址,因为f7()所带的参数不是按const引用传递的,所以它允许对临时对象X进行修改。但是编译器知道:一旦表达式计算结束,该临时对象也会不复存在,因此,对临时对象X所作的任何修改也将丢失。由于把所有的临时对象自动设为const,这种情况导致编译期间错误,因此这种错误不难发现。

然而,下面的表达式是合法的:

```
f5() = X(1);
f5().modify();
```

尽管它们可以编译通过,但实际上存在问题。f5()返回一个X对象,而且对编译器来说,要满足上面的表达式,它必须创建临时对象来保存返回值。于是,在这两个表达式中,临时对象也被修改,表达式被编译过之后,临时对象也将被清除。结果,丢失了所有的修改,从而代码可能存在问题——但是编译器不会有任何提示信息。对于用户来说,像这样的表达式很简单,他可以找出问题所在,但是,当事情变得复杂后,就可能在这方面出差错。

类对象常量是怎样保存起来的,将在本章的后面介绍。

#### 8.3.3 传递和返回地址

如果传递或返回一个地址(一个指针或一个引用),客户程序员去取地址并修改其初值是可能的。如果使这个指针或者引用成为const,就会阻止这类事的发生,这是非常重要的事情。事实上,无论什么时候传递一个地址给一个函数,都应该尽可能用const修饰它。如果不这样做,就不能以const指针参数的方式使用这个函数。

是否选择返回一个指向const的指针或者引用,取决于想让客户程序员用它干什么。下面这个例子表明了如何使用const指针作为函数参数和返回值:

```
//: C08:ConstPointer.cpp
// Constant pointer arg/return
```

```

void t(int*) {}

void u(const int* cip) {
    //! *cip = 2; // Illegal -- modifies value
    int i = *cip; // OK -- copies value
    //! int* ip2 = cip; // Illegal: non-const
}

const char* v() {
    // Returns address of static character array:
    return "result of function v()";
}

const int* const w() {
    static int i;
    return &i;
}

int main() {
    int x = 0;
    int* ip = &x;
    const int* cip = &x;
    t(ip); // OK
    //! t(cip); // Not OK
    u(ip); // OK
    u(cip); // Also OK
    //! char* cp = v(); // Not OK
    const char* ccp = v(); // OK
    //! int* ip2 = w(); // Not OK
    const int* const ccip = w(); // OK
    const int* cip2 = w(); // OK
    //! *w() = 1; // Not OK
} ///:~

```

函数`t()`把一个普通的非`const`指针作为一个参数，而函数`u()`把一个`const`指针作为参数。在函数`u()`里，会看到试图修改`const`指针所指的内容是非法的。当然，可以把信息拷贝进一个非`const`变量中。编译器也不允许使用存储在`const`指针里的地址来建立一个非`const`指针。

函数`v()`和`w()`测试返回值的语义。函数`v()`返回一个从字符数组的字面值中建立的`const char*`。在编译器建立了它并把它存储在静态存储区之后，这个声明实际上产生这个字符数组的字面值的地址。像前面提到的一样，从技术上讲，这字符数组是一个常量，这个常量由函数`v()`的返回值正确地表示。

`w()`的返回值要求这个指针及这个指针所指向的对象均为常量。像函数`v()`一样，仅仅因为它是静态的，所以在函数返回后由`w()`返回的值是有效的。函数不能返回指向局部栈变量的指针，这是因为在函数返回后它们就无效了，而且栈也被清除了。可返回的另一个普通指针是在堆中分配的存储地址，在函数返回后它仍然有效。

在`main()`中，函数被各种参数测试。函数`t()`将接受一个非`const`指针参数。但是，如果想传给它一个指向`const`的指针，那么将不能防止`t()`会丢下这个指针所指的内容不管，所以编译器会给出一个错误信息。函数`u()`带一个`const`指针，所以它接受两种类型的参数。这样，带`const`指针参数的函数比不带`const`指针参数的函数更具一般性。

正如所期望的一样，函数`v()`的返回值只可以被赋给一个`const`指针。编译器拒绝把函数`w()`的返回值赋给一个非`const`指针，而接受一个`const int* const`，但令人奇怪的是它也接受一个`const int*`，这不是与返回类型恰好匹配的。又正如前面所讲的，因为这个值（包含在指针中的地址）正被拷贝，所以自动保持这样的约定：原始变量不能被改变。因此，只有当把`const int*const`中的第二个`const`当做一个左值使用时（编译器会阻止这种情况），它才能显示其意义所在。

### 8.3.3.1 标准参数传递

在C语言中，按值传递是最常见的。当想传递地址时，惟一的选择就是使用指针<sup>①</sup>。然而，在C++中这两种方法都受重视。相反，当传递一个参数时，首先选择按引用传递，而且是`const`引用。对于客户程序员来说，这样做语法与按值传递是一样的，所以不会像使用指针那样的混淆——他们甚至不必考虑指针。对于函数的创建者来说，传递地址总比传递整个类对象更有效，如果按`const`引用来传递，意味着函数将不改变该地址所指的内容，从客户程序员的观点来看，效果就像按值传递一样（只是更有效）。

由于引用的语法（对于调用者它看起来像按值传递）的原因，把一个临时对象传递给接受`const`引用的函数是可能的，但不能把一个临时对象传递给接受指针的函数——对于指针，它必须明确地接受地址。所以，按引用传递会产生一个从来不会在C中出现的新的情形：一个总是`const`的临时变量，它的地址可以被传递给一个函数。这就是为什么当临时变量按引用传递给一个函数时，这个函数的参数必须是`const`引用的原因。下面的例子说明了这一点：

```
//: C08:ConstTemporary.cpp
// Temporaries are const

class X {};

X f() { return X(); } // Return by value

void g1(X&) {} // Pass by non-const reference
void g2(const X&) {} // Pass by const reference

int main() {
    // Error: const temporary created by f():
    //! g1(f());
    // OK: g2 takes a const reference:
    g2(f());
} ///:~
```

函数`f()`按值返回类`X`的一个对象。这意味着当立即取`f()`的返回值并把它传递给另外一个函数时（正如`g1()`和`g2()`函数的调用），将建立一个临时量，该临时量是`const`。这样，函数`g1()`中的调用是错误的，因为`g1()`不接受`const`引用，但是函数`g2()`中的调用是正确的。

## 8.4 类

本节介绍`const`用于类的两种办法。程序员可能想在一个类里建立一个局部`const`，将它用在常数表达式里，这个常数表达式在编译期间被求值。然而，`const`的意思在类里是不同的，所以为了创建类的`const`数据成员，必须了解这一选择。

① 有些人甚至会说C中的一切都是按值来传递，因为当传递一个指针时，也会得到了一份副本（所以是通过值传递指针的）。但我认为，无论这种看法有多么准确，它都会使这个问题变得更加混乱而不易理解。



还可以使整个对象作为**const**（正如刚刚看到的，编译器总是将临时类对象作为常量）。但是，要保持类对象为常量却比较复杂。编译器能保证一个内建类型为常量，但不能控制类中的复杂性。为了保证一个类对象为常量，引进了**const**成员函数：**const**成员函数只能对于**const**对象调用。

#### 8.4.1 类里的const

常数表达式使用常量的地方之一是在类里。典型的例子是在一个类里建立一个数组，并用**const**代替**#define**设置数组大小以及用于有关数组的计算。数组大小一直隐藏在类里，这样，如果用**size**表示数组大小，就可以把**size**这个名字用在另一个类里而不发生冲突。然而所有的**#define**从定义的地方起就被预处理器看成是全局的，所以用**#define**就不会得到预期的效果。

读者可能认为合乎逻辑的选择是把一个**const**放在类里。但这不会产生预期的结果。在一个类里，**const**又部分地恢复到它在C语言中的含义。它在每个类对象里分配存储并代表一个值，这个值一旦被初始化以后就不能改变。在一个类里使用**const**意味着“在这个对象生命周期内，它是一个常量”。然而，对这个常量来讲，每个不同的对象可以含有一个不同的值。

这样，在一个类里建立一个普通的（非**static**的）**const**时，不能给它初值。这个初始化工作必须在构造函数里进行，当然，要在构造函数的某个特别的地方进行。因为**const**必须在建立它的地方被初始化，所以在构造函数的主体里，**const**必定已被初始化了。否则，就只有等待，直到在构造函数主体以后的某个地方给它初始化，这意味着过一会儿才给**const**初始化。当然，无法防止在构造函数主体的不同地方改变**const**的值。

##### 8.4.1.1 构造函数初始化列表

在构造函数里有个专门初始化的地方，这就是构造函数初始化列表（*constructor initializer list*），起初用在继承里（继承将在第14章介绍）。构造函数初始化表列表（顾名思义，只出现在构造函数的定义里）是一个出现在函数参数表和冒号后，但在构造函数主体开头的花括号前的“函数调用列表”。这提醒人们，表里的初始化发生在构造函数的任何代码执行之前。这是初始化所有**const**的地方，所以类里的**const**的正确形式是：

```
//: C08:ConstInitialization.cpp
// Initializing const in classes
#include <iostream>
using namespace std;

class Fred {
    const int size;
public:
    Fred(int sz);
    void print();
};

Fred::Fred(int sz) : size(sz) {}
void Fred::print() { cout << size << endl; }

int main() {
    Fred a(1), b(2), c(3);
    a.print(), b.print(), c.print();
} ///:~
```



开始时，上面显示的构造函数初始化列表的形式容易使人们混淆，因为人们不习惯把一个内建类型看成好像也有一个构造函数。

#### 8.4.1.2 内建类型的“构造函数”

随着语言的发展以及人们为使用户定义类型看起来像内建类型一样所作的努力，有时似乎使内建数据类型看起来像用户定义类型更好。在构造函数初始化列表里，可以把一个内建类型看成好像它有一个构造函数，就像下面这样：

```
//: C08:BuiltInTypeConstructors.cpp
#include <iostream>
using namespace std;

class B {
    int i;
public:
    B(int ii);
    void print();
};

B::B(int ii) : i(ii) {}
void B::print() { cout << i << endl; }

int main() {
    B a(1), b(2);
    float pi(3.14159);
    a.print(); b.print();
    cout << pi << endl;
} ///:~
```

这在初始化**const**数据成员时尤为关键，因为它们必须进入函数体前被初始化。

我们还可以把这个内建类型的“构造函数”（仅指赋值）扩展为一般的情形，这就是为什么要在上段代码中加入**float pi (3.14159)**定义的原因。

把一个内建类型封装在一个类里以保证用构造函数初始化，这是很有用的。例如，下面是一个**Integer**类：

```
//: C08:EncapsulatingTypes.cpp
#include <iostream>
using namespace std;

class Integer {
    int i;
public:
    Integer(int ii = 0);
    void print();
};

Integer::Integer(int ii) : i(ii) {}
void Integer::print() { cout << i << ' '; }

int main() {
    Integer i[100];
    for(int j = 0; j < 100; j++)
```



```
    i[j].print();
} ///:~
```

在`main()`中的**Integer**数组元素都自动地初始化为零。与**for**循环和**memset()**相比,这种初始化并不必付出更多的开销。很多编译器可以很容易地把它优化成一个很快的过程。

#### 8.4.2 编译期间类里的常量

上面所使用的**const**是有趣的,也可能很有用,但是它没有解决最初的问题,这就是:如何让一个类有编译期间的常量成员?这就要求使用另外一个关键字**static**,在第10章才会对它进行详尽的介绍。在这种情形下,关键字**static**意味着“不管类的对象被创建多少次,都只有一个实例”,这正是所需要的:类中的一个常量成员,在该类的所有对象中它都一样。因此,一个内建类型的**static const**可以看做一个编译期间的常量。

必须在**static const**定义的地方对它进行初始化。这是在类中使用**static const**的特征之一,也显得有点与众不同。这种情况只会伴随**static const**一起出现:也许更喜欢把它用在其他情况下,但不行,因为所有其他的数据成员也必须在构造函数或其他成员函数里初始化。

下面有一个例子,它说明了在一个类里创建和使用一个叫做**size**的**static const**,这个类表示一个存放字符串指针的栈<sup>①</sup>。

```
//: C08:StringStack.cpp
// Using static const to create a
// compile-time constant inside a class
#include <string>
#include <iostream>
using namespace std;

class StringStack {
    static const int size = 100;
    const string* stack[size];
    int index;
public:
    StringStack();
    void push(const string* s);
    const string* pop();
};

StringStack::StringStack() : index(0) {
    memset(stack, 0, size * sizeof(string*));
}

void StringStack::push(const string* s) {
    if(index < size)
        stack[index++] = s;
}

const string* StringStack::pop() {
    if(index > 0) {
        const string* rv = stack[--index];
        stack[index] = 0;
    }
}
```

① 在写本书时,并不是所有的编译器都支持该特征。



```

        return rv;
    }
    return 0;
}

string iceCream[] = {
    "pralines & cream",
    "fudge ripple",
    "jamocha almond fudge",
    "wild mountain blackberry",
    "raspberry sorbet",
    "lemon swirl",
    "rocky road",
    "deep chocolate fudge"
};

const int iCsz =
    sizeof iceCream / sizeof *iceCream;

int main() {
    StringStack ss;
    for(int i = 0; i < iCsz; i++)
        ss.push(&iceCream[i]);
    const string* cp;
    while((cp = ss.pop()) != 0)
        cout << *cp << endl;
} ///:~

```

因为`size`用来决定数组`stack`的大小，所以，它实际上是一个编译期间常量，但隐藏在类中。

注意`push()`带有一个`const string*`参数，`pop()`返回一个`const string*`，`StringStack`保存`const string*`。否则，就不能用`StringStack`存放在`iceCream`中的指针。可是，它阻止程序员做改变包含在`StringStack`中的对象的任何事情。当然，这种限制不是普遍存在的。

#### 8.4.2.1 旧代码中的“enum hack”

在旧版本的C++中，不支持在类中使用`static const`。这意味着`const`对在类中的常量表达式不起作用，不过，人们还是想做到这一点。于是，一个典型的解决办法就是使用不带实例的无标记`enum`（通常称为“enum hack”）。一个枚举在编译期间必须有值，它在类中局部出现，而且它的值对于常量表达式是可以使用的。所以有下面的代码：

```

//: C08:EnumHack.cpp
#include <iostream>
using namespace std;

class Bunch {
    enum { size = 1000 };
    int i[size];
};

int main() {
    cout << "sizeof(Bunch) = " << sizeof(Bunch)
        << ", sizeof(i[1000]) = "
        << sizeof(int[1000]) << endl;
} ///:~

```



这里使用的**enum**保证不占用对象的存储空间，编译期间得到枚举值。也可以明确地给枚举元素赋值。如下所示：

```
enum { one = 1, two = 2, three };
```

在一个完整的**enum**类型中，要是没有为枚举元素特别指定值的话，编译器会从最近的值开始计算，例如上面的**three**的值为3。

在上面**StringStack.cpp**中，可以把

```
static const int size = 100;
```

替换为

```
enum { size = 100 };
```

虽然会经常在以前的程序代码里看到使用**enum**技术，但在C++中增加了**static const**特性，正是为了解决这个问题。但是，没有绝对的理由说明一定要优先选择**static const**而尽量不用**enum hack**，本书使用**enum hack**是由于写这本书的时候大多数的编译器都支持这种特性。

### 8.4.3 const对象和成员函数

可以用**const**限定类成员函数，这是什么意思呢？为了搞清楚这一点，必须首先掌握**const**对象的概念。

用户定义类型和内建类型一样，都可以定义一个**const**对象。例如：

```
const int i = 1;
const blob b(2);
```

这里，**b**是类型**blob**的一个**const**对象。它的构造函数被调用，且其参数为“2”。由于编译器强调对象为**const**的，因此它必须保证对象的数据成员在其生命期内不被改变。它可以很容易地保证公有数据不被改变，但是它怎么知道哪些成员函数将会改变数据？它又如何知道哪些成员函数对于**const**对象来说是“安全”的呢？

如果声明一个成员函数为**const**，则等于告诉编译器该成员函数可以为一个**const**对象所调用。一个没有被明确声明为**const**的成员函数被看成是将要修改对象中数据成员的函数，而且编译器不允许它为一个**const**对象所调用。

然而，不能到此为止。仅仅声明一个函数在类定义里是**const**的，还不能保证成员函数按声明的方式去做，所以编译器强迫程序员在定义函数时要重申**const**说明。（**const**已成为函数识别符的一部分，所以编译器和连接程序都要检查**const**。）为确保函数定义的常量性，如果我们改变对象中的任何成员或调用一个非**const**成员函数，编译器就将发出一个出错信息，这样，可以保证声明为**const**的任何成员函数能够按定义方式运行。

要理解声明**const**成员函数的语法，首先注意前面的带**const**的函数声明，它表示函数的返回值是**const**，但这不会产生想要的结果。相反，必须把修饰符**const**放在函数参数表的后面，例如：

```
//: C08:ConstMember.cpp
class X {
    int i;
public:
    X(int ii);
```

```

    int f() const;
};

X::X(int ii) : i(ii) {}
int X::f() const { return i; }

int main() {
    X x1(10);
    const X x2(20);
    x1.f();
    x2.f();
} ///:~

```

关键字**const**必须用同样的方式重复出现在定义里，否则编译器把它看成一个不同的函数，因为**f()**是一个**const**成员函数，所以不管它试图以何种方式改变**i**或者调用另一个非**const**成员函数，编译器都把它标记成一个错误。

一个**const**成员函数调用**const**和非**const**对象是安全的，因此，可以把它看做成员函数的最一般形式（不幸的是，成员函数并不会自动地默认为**const**）。不修改数据成员的任何函数都应该把它们声明为**const**，这样它可以和**const**对象一起使用。

下面是一个比较**const**和非**const**成员函数的例子：

```

//: C08:Quoter.cpp
// Random quote selection
#include <iostream>
#include <cstdlib> // Random number generator
#include <ctime> // To seed random generator
using namespace std;

class Quoter {
    int lastquote;
public:
    Quoter();
    int lastQuote() const;
    const char* quote();
};

Quoter::Quoter() {
    lastquote = -1;
    srand(time(0)); // Seed random number generator
}

int Quoter::lastQuote() const {
    return lastquote;
}

const char* Quoter::quote() {
    static const char* quotes[] = {
        "Are we having fun yet?",
        "Doctors always know best",
        "Is it ... Atomic?",
        "Fear is obscene",
        "There is no scientific evidence "
        "to support the idea "
        "that life is serious",
    }
}

```



```

        "Things that make us happy, make us wise",
    };
    const int qsize = sizeof quotes/sizeof *quotes;
    int qnum = rand() % qsize;
    while(lastquote >= 0 && qnum == lastquote)
        qnum = rand() % qsize;
    return quotes[lastquote = qnum];
}

int main() {
    Quoter q;
    const Quoter cq;
    cq.lastQuote(); // OK
    //! cq.quote(); // Not OK; non const function
    for(int i = 0; i < 20; i++)
        cout << q.quote() << endl;
} ///:~

```

构造函数和析构函数都不是**const**成员函数，因为它们在初始化和清除时，总是对对象作些修改。**quote()**成员函数也不能是**const**函数，因为它要修改数据成员**lastquote**（请看**return**语句）。而**lastQuote()**没做修改，所以它可以成为**const**函数，而且也可以被**const**对象**cq**安全地调用。

#### 8.4.3.1 可变的：按位**const**和按逻辑**const**

如果想要建立一个**const**成员函数，但仍然想在对象里改变某些数据，这时该怎么办呢？这关系到按位（*bitwise*）**const**和按逻辑（*logical*）**const**（有时也称为按成员（*memberwise*）**const**）的区别。按位**const**意思是对象中的每个字节都是固定的，所以对象的每个位映像从不改变。按逻辑**const**意思是，虽然整个对象从概念上讲是不变的，但是可以以成员为单位改变。当编译器被告知一个对象是**const**对象时，它将绝对保护这个对象按位的常量性。要实现按逻辑**const**的属性，有两种由内部**const**成员函数改变数据成员的方法。

第一种方法已成为过去，称为“强制转换常量性（*casting away constness*）”。它以相当奇怪的方式执行。取**this**（这个关键字产生当前对象的地址）并把强制转换成指当前类型对象的指针。看来**this**已经是所需的指针，但是，在**const**成员函数内部，它实际上是一个**const**指针，所以，还应把它强制转换成一个普通指针，这样就可以在那个运算中去掉常量性。下面是一个例子：

```

//: C08:Castaway.cpp
// "Casting away" constness

class Y {
    int i;
public:
    Y();
    void f() const;
};

Y::Y() { i = 0; }

void Y::f() const {
    //! i++; // Error -- const member function
    ((Y*)this)->i++; // OK: cast away const-ness
}

```



```

    // Better: use C++ explicit cast syntax:
    (const_cast<Y*>(this))->i++;
}

int main() {
    const Y yy;
    yy.f(); // Actually changes it!
} ///:~

```

这种方法是可行的，在过去的程序代码里可以看到这种用法，但这不是首选的技术。问题是：常量性的缺乏隐藏在成员函数的定义中，并且没有来自类接口的线索知道对象的数据实际上被修改，除非用户不能见到源代码（用户必然怀疑常量性被转换了，并寻找这一类型转换）。为了公开这一切，应当在类声明里使用关键字**mutable**，以指定一个特定的数据成员可以在一个**const**对象里被改变。

```

//: C08:Mutable.cpp
// The "mutable" keyword

class Z {
    int i;
    mutable int j;
public:
    Z();
    void f() const;
};

Z::Z() : i(0), j(0) {}

void Z::f() const {
    //! i++; // Error -- const member function
    j++; // OK: mutable
}

int main() {
    const Z zz;
    zz.f(); // Actually changes it!
} ///:~

```

现在，类用户可从声明里看到哪个成员能够用**const**成员函数进行修改。

#### 8.4.3.2 只读存储能力

如果一个对象被定义成**const**对象，它就成为被放进只读存储器（ROM）中的候选者，这经常是嵌入式系统程序设计中要考虑做的重要事情。然而，只建立一个**const**对象是不够的——只读存储能力所需要的条件要严格得多。当然，这个对象还应是按位**const**的，而不是按逻辑**const**的。如果只通过关键字**mutable**实现按逻辑常量化的话，就容易看出这一点。如果在一个**const**成员函数里的**const**被强制转换了，编译器可能检测不到这种情况。另外：

- 1) **class**或**struct**必须没有用户定义的构造函数或析构函数。
- 2) 这里不能有基类（将在第14章中谈到），也不能包含有用户定义构造函数或析构函数的成员对象。

在只读存储能力类型的**const**对象中的任何部分上，有关写操作的影响没有定义。虽然适当形成式的对象可被放进ROM里，但是目前还没有什么对象需要放进ROM里。



## 8.5 volatile

**volatile**的语法与**const**是一样的，但是**volatile**的意思是“在编译器认识的范围外，这个数据可以被改变”。不知何故，环境正在改变数据（可能通过多任务、多线程或者中断处理），所以，**volatile**告诉编译器不要擅自作出有关该数据的任何假定，优化期间尤其如此。

如果编译器说：“我已经把数据读进寄存器，而且再没有与寄存器接触”。一般情况下，它不需要再读这个数据。但是，如果数据是**volatile**修饰的，编译器就不能作出这样的假定，因为这个数据可能被其他进程改变了，它必须重读这个数据而不是优化这个代码来消除通常情况下那些冗余的读操作代码。

就像建立**const**对象一样，程序员也可以建立**volatile**对象，甚至还可以建立**const volatile**对象，这个对象不能被客户程序员改变，但可通过外部的代理程序改变。下面的例子描述了一个类，这个类涉及通信硬件：

```
//: C08:Volatile.cpp
// The volatile keyword

class Comm {
    const volatile unsigned char byte;
    volatile unsigned char flag;
    enum { bufsize = 100 };
    unsigned char buf[bufsize];
    int index;
public:
    Comm();
    void isr() volatile;
    char read(int index) const;
};

Comm::Comm() : index(0), byte(0), flag(0) {}

// Only a demo; won't actually work
// as an interrupt service routine:
void Comm::isr() volatile {
    flag = 0;
    buf[index++] = byte;
    // Wrap to beginning of buffer:
    if(index >= bufsize) index = 0;
}

char Comm::read(int index) const {
    if(index < 0 || index >= bufsize)
        return 0;
    return buf[index];
}

int main() {
    volatile Comm Port;
    Port.isr(); // OK
    //! Port.read(0); // Error, read() not volatile
} ///:~
```

就像**const**一样，我们可以对数据成员、成员函数和对象本身使用**volatile**，可以对**volatile**

对象调用**volatile**成员函数。

函数**isr()**不能像中断服务程序那样使用的原因是：在一个成员函数里，当前对象（**this**）的地址必须被秘密地传递，而中断服务程序**ISR**一般根本不要参数。为解决这个问题，可以让**isr()**是静态成员函数，这是第10章讨论的主题。

**volatile**的语法与**const**是一样的，所以对它们的讨论经常被放在一起。为指明可以选择两个中的任何一个，把它们连在一起通称为**c-v**限定词（*c-v qualifier*）。

## 8.6 小结

关键字**const**能将对象、函数参数、返回值和成员函数定义为常量，并能消除预处理器的值替代而不使预处理器的影响。所有这些都为程序设计提供了又一种非常好的类型检查形式以及安全性。使用所谓的常量正确性（*const correctness*）（在任何可能的地方使用**const**）已成为项目的救星。

尽管可以忽视**const**而继续使用旧的C代码习惯，但是它确实有帮助，第11章将改变他们的做法，在第11章中将开始大量使用引用，那时将看到对函数参数使用**const**是多么关键。

## 8.7 练习

部分练习题的答案可以在本书的电子文档“*Annotated Solution Guide for Thinking in C++*”中找到，只需支付很少的费用就可以从<http://www.BruceEckel.com>得到这个电子文档。

- 8-1 创建三个**const int**值，把它们加到一起得到一个值用来在一个数组定义中决定该数组的大小。在C中编译一遍相同的代码，看看会出现什么情况（通过使用命令行标记，可以将C++编译器改作为C编译器运行）。
- 8-2 自行证实C编译器和C++编译器对于**const**的处理是不同的。创建一个全局的**const**并将它用在一個全局的常量表达式中；然后分别用C和C++编译它。
- 8-3 为所有的内建类型创建**const**定义及其变量。和其他的**const**一起在表达式中使用定义新的**const**。并确保编译正确无误。
- 8-4 在一个头文件中创建一个**const**定义，包含这个头文件在两个.cpp文件中，然后编译这些文件并连接它们。要保证正确无误，再在C环境下试一遍。
- 8-5 创建一个**const**，当程序运行时，通过读时间决定它的值（必须使用标准的头文件<ctime>），然后在这个程序中读时间的第二个值，并赋给**const**，看看会有什么结果。
- 8-6 创建一个**char**的**const**数组，然后尝试修改**string**数组中的某一个值。
- 8-7 在一个文件中创建一个**extern const**声明，该文件的**main()**函数打印**extern const**的值，在另外一个文件中定义**extern const**，然后编译和连接这两个文件。
- 8-8 使用不同的声明形式创建两个指向**const long**的指针，一个指针指向一个**long**数组。演示能让指针增加和减少，但不能改变它所指向的值。
- 8-9 写一个指向**double**类型的**const**指针，让它指向**double**数组。显示能改变指针指向的内容，但不能增加或减小指针。
- 8-10 写一个指向**const**对象的**const**指针。显示只能读指针所指向的值，但不能改变该指针或它所指向的值。
- 8-11 删除**PointerAssignment.cpp**文件中代码的错误行前的注释，看看编译器会产生什么样

的错误。

- 8-12 创建一个字符数组字面值和一个指向该数组开始点的指针，使用这个指针修改数组中的元素，看看编译器是否会报告出错，应当出错吗？如果没有，为什么会认为出错？
- 8-13 创建一个函数，它带有一个以**const**值传递的参数，然后在函数体中试图改变该参数。
- 8-14 创建一个函数，它带有一个按值传递的**float**参数。在函数体中，把**const float&**绑定到函数的参数上，并且从那时起仅仅使用引用，以确保不改变参数。
- 8-15 修改**ConstReturnValues.cpp**文件，每次删除错误行前的注释，看看编译器会产生什么错误信息。
- 8-16 修改**ConstPointer.cpp**文件，每次删除错误行前的注释，看看编译器会产生什么错误信息。
- 8-17 制造文件**ConstPointer.cpp**的新版，名为**ConstReference.cpp**，其中把前者使用的指针用引用代替（也许需要用到第11章中的知识）。
- 8-18 修改**ConstTemporary.cpp**文件，删除错误行前的注释，看看编译器会产生什么错误信息。
- 8-19 创建一个包含**const**和非**const float**成员的类。用构造函数的初始化列表进行初始化。
- 8-20 创建类**MyString**，它包含一个**string**成员、一个初始化该**string**成员的构造函数以及**print()**函数。修改**StringStack.cpp**文件，以便让容器保存**MyString**对象，**main()**函数打印它们。
- 8-21 创建包含一个**const**成员和一个枚举成员的类。在构造函数的初始化列表中初始化**const**成员，无标记的枚举成员用来决定数组大小。
- 8-22 在**ConstMember.cpp**文件中，删除成员函数定义前的**const**限定符，但是让**const**限定符出现在声明中，看看会得到何种类型的编译器错误信息。
- 8-23 创建一个类，它有一个**const**和非**const**成员函数。再创建该类的**const**和非**const**对象，用不同类型的对象调用不同类型的成员函数。
- 8-24 创建一个类，它有一个**const**和非**const**成员函数。尝试从**const**成员函数中调用非**const**成员函数，看看会得到何种类型的编译器错误消息。
- 8-25 在**Mutable.cpp**文件中，删除错误行前的注释，看看编译器会产生什么错误信息。
- 8-26 修改**Quoter.cpp**文件的函数**quote()**，使它变为**const**成员函数和**mutable**。
- 8-27 创建一个类，它有一个**volatile**数据成员，创建一个**volatile**和一个非**volatile**成员函数用于修改**volatile**数据成员。看看编译器会出现什么情况。创建该类的**volatile**和非**volatile**对象，尝试调用**volatile**和非**volatile**成员函数，看看哪一个调用会成功，哪一个调用不成功，以及编译器会产生什么样的错误信息。
- 8-28 创建一个具有成员函数**fly()**的名为**bird**的类和一个不含**fly()**的名为**rock**的类。建立一个**rock**对象，取它的地址，并把它赋给一个**void\***。再取这个**void\***，把它赋给一个**bird\***（必须使用类型转换），通过指针调用函数**fly()**。为什么C语言允许通过**void\***（而不是类型转换）公开地赋值？这是C语言中的一个“缺陷”吗？不能把它推广到C++中吗？

## 内联函数

C++从C中继承的一个重要特征是效率。假如C++的效率显著地低于C的效率，那么就会有很大一批程序员不去使用它。

在C中，保持效率的一个方法是使用宏(*macro*)。宏可以不要普通的函数调用代价就可使之看起来像函数调用。宏的实现是用预处理器而不是编译器。预处理器直接用宏代码代替宏调用，所以就没有了参数压栈、生成汇编语言的CALL、返回参数、执行汇编语言的RETURN等的开销。所有的工作由预处理器来完成，因此不用花费什么就具有了程序调用的便利和可读性。

在C++中，使用预处理器宏存在两个问题。第一个问题在C中也存在：宏看起来像一个函数调用，但并不总是这样。这样就隐藏了难以发现的错误。第二个问题是C++特有的：预处理器不允许访问类的成员数据。这意味着预处理器宏不能用作类的成员函数。

为了既保持预处理器的效率又增加安全性，而且还能像一般成员函数一样可以在类里访问自如，C++引入了内联函数(*inline function*)。本章将介绍C++中预处理器宏存在的问题、在C++中如何用内联函数解决这些问题以及使用内联函数的方针和内联函数的工作机制。

### 9.1 预处理器的缺陷

预处理器宏存在问题的关键是我们可能认为预处理器的行为和编译器的行为一样。当然，这是有意使宏在外观上和行为上与函数调用一样，因此容易被混淆。当微妙的差异出现时，问题就出现了。

考虑下面这个简单例子：

```
#define F (x) (x + 1)
```

现在假如有一个如下所示的F调用：

```
F(1)
```

预处理器展开它，出现下面不希望的情况：

```
(x) (x + 1) (1)
```

出现这个问题是因为在宏定义中F和括号之间存在空格。当这个空格取消后，调用宏时可以有空格空隙。像下面的调用：

```
F(1)
```

依然可以正确地展开为：

```
(1 + 1)
```

上面的例子虽然非常微不足道，但问题非常明显。当在宏调用中使用表达式作为参数时，真正的问题就出现了。

这里存在两个问题。第一个问题是表达式在宏内展开，所以它们的优先级不同于所期望的优先级。例如：

```
#define FLOOR(x,b) x>=b?0:1
```

现在假如用表达式作参数：

```
if(FLOOR(a&0x0f,0x07)) // ...
```

宏将展开成：

```
if(a&0x0f>=0x07?0:1)
```

因为&的优先级比>=的低，所以宏的展开结果将会使我们惊讶。一旦发现这个问题，可以通过在宏定义内的各个地方使用括弧来解决。（这是创建预处理器宏时使用的好方法。）上面的定义可改写成如下：

```
#define FLOOR(x,b) ((x)>=(b)?0:1)
```

然而，发现问题可能很难，我们可能一直认为宏的行为是正确的。在前面没有加括号的版本的例子中，大多数表达式将正确工作，因为>=的优先级比像+、/、--，甚至按位移动操作符的优先级都低。因此，很容易想到它对于所有的表达式都正确，包括那些位逻辑操作符。

前面的问题可以通过谨慎地编程来解决：在宏中将所有的内容都用括号括起来。第二个问题则复杂一些。不像普通函数，每次在宏中使用一个参数，都对这个参数求值。只要使用普通变量调用宏，求值就无危险。但假如参数求值有副作用，那么结果可能出乎预料，并肯定不能模仿函数行为。

例如，下面这个宏决定它的参数是否在一定范围：

```
#define BAND(x) (((x)>5 && (x)<10) ? (x) : 0)
```

只要使用一个“普通”参数，宏和真的函数的工作方式非常相似。但只要一松懈并开始相信它是一个真的函数时，问题就出现了。如下所示：

```
//: C09:MacroSideEffects.cpp
#include "../require.h"
#include <fstream>
using namespace std;

#define BAND(x) (((x)>5 && (x)<10) ? (x) : 0)

int main() {
    ofstream out("macro.out");
    assure(out, "macro.out");
    for(int i = 4; i < 11; i++) {
        int a = i;
        out << "a = " << a << endl << '\t';
        out << "BAND(++a)=" << BAND(++a) << endl;
        out << "\t a = " << a << endl;
    }
} ///:~
```

注意宏名中所有大写字母的使用。这是一种很有用的做法，因为大写的字母告诉读者这是一个宏而不是一个函数，所以如果出现问题，也可以起到一定的提示作用。

下面是这个程序的输出，它完全不是想从真正的函数期望得到的结果：

```

a = 4
BAND(++a)=0
a = 5
a = 5
BAND(++a)=8
a = 8
a = 6
BAND(++a)=9
a = 9
a = 7
BAND(++a)=10
a = 10
a = 8
BAND(++a)=0
a = 10
a = 9
BAND(++a)=0
a = 11
a = 10
BAND(++a)=0
a = 12

```

当`a`等于4时，仅测试了条件表达式第一部分，表达式只求值一次，所以宏调用的副作用是`a`等于5，这是在相同的情况下从普通函数调用所期望得到的。但当数字在值域范围内时，两个表达式都测试，产生两次自增操作。产生这个结果是由于再次对参数操作。一旦数字出了范围，两个条件仍然测试，所以也产生两次自增操作。根据参数不同产生的副作用也不同。

很清楚，这不是我们想从看起来像函数调用的宏中所希望得到的行为。在这种情况下，明显的解决方法是设计真正的函数。当然，如果多次调用函数将会增加额外的开销并可能降低效率。不幸的是，问题可能并不总是如此明显。可能不知不觉地得到一个包含混合函数和宏的库函数，所以像这样的问题可能隐藏了一些难以发现的错误。例如，在`cstdio`中的`putc()`宏可能对它的第二个参数求值两次。这在标准C中作了详细说明。作为宏`toupper()`不谨慎地执行也会对第二个参数求值多次。如在使用`toupper(*p++)`<sup>①</sup>时会产生不希望的结果。

### 9.1.1 宏和访问

当然，在C中需要对预处理器宏谨慎地编码和使用。要不是因为宏没有成员函数作用域这一要求，我们也会在C++中侥幸成功地使用它。预处理器只是简单地执行字符替代，所以不可能用下面这样或近似的形式写：

```

class X {
    int i;
public:
    #define VAL(X::i) // Error

```

另外，这里没有指明正在使用哪个对象。在宏里简直没有办法表示类的范围。由于没有可以取代预处理器宏的方法，程序设计者出于效率考虑，不得不让一些数据成员成为`public`类型，这样就会暴露内部实现并妨碍在这个实现中的改变，从而消除了`private`提供的保护。

① 更多的细节参见Andrew Koenig 的著作《C Traps & Pitfalls》(Addison-Wesley, 1989)。

## 9.2 内联函数

在解决C++中宏访问**private**类成员的问题过程中，所有和预处理器宏有关的问题也随之排除了。这是通过使宏被编译器控制来实现的。在C++中，宏的概念是作为内联函数（*inline function*）来实现的，而内联函数无论从那一方面上说都是真正的函数。内联函数能够像普通函数一样具有我们所有期望的任何行为。惟一不同之处是内联函数在适当的地方像宏一样展开，所以不需要函数调用的开销。因此，应该（几乎）永远不使用宏，只使用内联函数。

任何在类中定义的函数自动地成为内联函数，但也可以在非类的函数前面加上**inline**关键字使之成为内联函数。但为了使之有效，必须使函数体和声明结合在一起，否则，编译器将它作为普通函数对待。因此

```
inline int plusOne(int x);
```

没有任何效果，仅仅只是声明函数（这不一定能够在稍后某个时候得到一个内联定义）。成功的方法如下：

```
inline int plusOne(int x) { return ++x; }
```

注意，编译器将检查函数参数列表使用是否正确，并返回值（进行必要的转换）。这些事情是预处理器无法完成的。假如对于上面的内联函数写成一个预处理器宏的话，将得到不想要的副作用。

一般应该把内联定义放在头文件里。当编译器看到这个定义时，它把函数类型（函数名+返回值）和函数体放到符号表里。当使用函数时，编译器检查以确保调用是正确的且返回值被正确使用，然后将函数调用替换为函数体，因而消除了开销。内联代码的确占用空间，但假如函数较小，这实际上比为了一个普通函数调用而产生的代码（参数压栈和执行CALL）占用的空间还少。

在头文件中，内联函数处于一种特殊状态，因为在头文件中声明该函数，所以必须包含头文件和该函数的定义，这些定义在每个用到该函数的文件中，但是不会出现产生多个定义错误的情况（不过，在任何使用内联函数地方该内联函数的定义都必须是相同的）。

### 9.2.1 类内部的内联函数

为了定义内联函数，通常必须在函数定义前面放一个**inline**关键字。但这在类内部定义内联函数时并不是必须的。任何在类内部定义的函数自动地成为内联函数。如下例：

```
//: C09:Inline.cpp
// Inlines inside classes
#include <iostream>
#include <string>
using namespace std;

class Point {
    int i, j, k;
public:
    Point(): i(0), j(0), k(0) {}
    Point(int ii, int jj, int kk)
        : i(ii), j(jj), k(kk) {}
    void print(const string& msg = "") const {
```

```

        if(msg.size() != 0) cout << msg << endl;
        cout << "i = " << i << ", "
              << "j = " << j << ", "
              << "k = " << k << endl;
    }
};

int main() {
    Point p, q(1,2,3);
    p.print("value of p");
    q.print("value of q");
} ///:~

```

两个构造函数和`print()`函数都默认为内联函数。注意在`main()`函数中使用内联函数是自然而然的事。一个函数的逻辑行为必须相同（要不然会出现编译错误），不管它是否是内联函数，我们就会看到，惟一不同之处在于它们的效率不一样。

当然，因为类内部的内联函数节省了在外部定义成员函数的额外步骤，所以我们一定想在类声明内每一处都使用内联函数。但应记住，使用内联函数的目的是减少函数调用的开销。但是，假如函数较大，由于需要在调用函数的每一处重复复制代码，这样将使代码膨胀，在速度方面获得的好处就会减少（惟一可靠的办法就是在程序上试验，看看使用内联函数的效果如何）。

### 9.2.2 访问函数

在类中内联函数的最重要的使用之一是用做访问函数（*access function*）。这是一个小函数，它容许读或修改对象状态——一个或几个内部变量。从下面的例子中，可以看访问函数为内联函数的原因。

```

//: C09:Access.cpp
// Inline access functions

class Access {
    int i;
public:
    int read() const { return i; }
    void set(int ii) { i = ii; }
};

int main() {
    Access A;
    A.set(100);
    int x = A.read();
} ///:~

```

这里，在类的设计者控制下，将类里面状态变量设计为私有，类的使用者就永远不会直接和它们发生联系了。对私有数据成员的所有访问只能通过成员函数接口进行。而且，这种访问是相当有效的。例如对于函数`read()`，若没用内联函数，对`read()`调用产生的代码将包括对`this`压栈和执行汇编语句`CALL`。对于大多数机器，产生的代码将比内联函数产生的代码大一些，执行的时间肯定要长。

不用内联函数，考虑效率的类设计者将忍不住简单地使`i`为公共成员，从而通过让用户直



接访问*i*来消除开销。从设计的角度看，这是很不好的。因为*i*将成为公共接口的一部分，所以意味着类设计者决不能修改它。我们将和称为*i*的一个*int*类型变量打交道。这是一个问题，因为可能在稍后觉得用一个*float*变量比用一个*int*变量代表状态信息更有用一些，但因为*int i*是公共接口的一部分，所以不能改变它。同样，想在读或是设置*i*值时执行加法运算也是不允许的，另一方面，假如总是使用成员函数读和修改一个对象的状态信息，那么就可以满意地修改对象内部一些描述。

另外，使用成员函数控制数据成员的访问允许在成员函数中增加代码以检测数据什么时候改变。这在程序调试时非常有用。如果数据成员是**public**的，任何人就可以任意改变它的值。

#### 9.2.2.1 访问器和修改器

一些人进一步把访问函数的概念分成访问器 (*accessor*) (用于从一个对象读状态信息) 和修改器 (*mutator*) (用于修改状态信息)。而且，可以用重载函数为访问器和修改器提供相同函数名，调用函数的方式决定了是读还是修改状态信息。

```
//: C09:Rectangle.cpp
// Accessors & mutators

class Rectangle {
    int wide, high;
public:
    Rectangle(int w = 0, int h = 0)
        : wide(w), high(h) {}
    int width() const { return wide; } // Read
    void width(int w) { wide = w; } // Set
    int height() const { return high; } // Read
    void height(int h) { high = h; } // Set
};

int main() {
    Rectangle r(19, 47);
    // Change width & height:
    r.height(2 * r.width());
    r.width(2 * r.height());
} ///:~
```

构造函数使用构造函数初始化列表 (这在第7章中做了简介，在第14章中将做详细介绍) 来初始化**wide**和**high**值 (对于内建数据类型使用伪构造函数调用形式)。

不能让成员函数名与数据成员名相同，于是我们也许想用下划线作为标识符的第一字符来区分这些数据成员。然而，第一个字符为下划线的标识符是保留的，所以不应该使用它们。

可以选用“get”和“set”来标识访问器和修改器。

```
//: C09:Rectangle2.cpp
// Accessors & mutators with "get" and "set"

class Rectangle {
    int width, height;
public:
    Rectangle(int w = 0, int h = 0)
        : width(w), height(h) {}
    int getWidth() const { return width; }
    void setWidth(int w) { width = w; }
```

```

    int getHeight() const { return height; }
    void setHeight(int h) { height = h; }
};

int main() {
    Rectangle r(19, 47);
    // Change width & height:
    r.setHeight(2 * r.getWidth());
    r.setWidth(2 * r.getHeight());
} ///:~

```

当然，访问器和修改器对于内部变量来说，不必是简单的管道。有时，它们可以执行一些比较复杂的计算。下面的例子使用标准的C库函数中的时间函数来生成简单的**Time**类：

```

///: C09:Cpptime.h
// A simple time class
#ifdef CPPTIME_H
#define CPPTIME_H
#include <ctime>
#include <cstring>

class Time {
    std::time_t t;
    std::tm local;
    char asciiRep[26];
    unsigned char lflag, aflag;
    void updateLocal() {
        if(!lflag) {
            local = *std::localtime(&t);
            lflag++;
        }
    }
    void updateAscii() {
        if(!aflag) {
            updateLocal();
            std::strcpy(asciiRep, std::asctime(&local));
            aflag++;
        }
    }
public:
    Time() { mark(); }
    void mark() {
        lflag = aflag = 0;
        std::time(&t);
    }
    const char* ascii() {
        updateAscii();
        return asciiRep;
    }
    // Difference in seconds:
    int delta(Time* dt) const {
        return int(std::difftime(t, dt->t));
    }
    int daylightSavings() {
        updateLocal();
        return local.tm_isdst;
    }
};

```



```

    }
    int dayOfYear() { // Since January 1
        updateLocal();
        return local.tm_yday;
    }
    int dayOfWeek() { // Since Sunday
        updateLocal();
        return local.tm_wday;
    }
    int since1900() { // Years since 1900
        updateLocal();
        return local.tm_year;
    }
    int month() { // Since January
        updateLocal();
        return local.tm_mon;
    }
    int dayOfMonth() {
        updateLocal();
        return local.tm_mday;
    }
    int hour() { // Since midnight, 24-hour clock
        updateLocal();
        return local.tm_hour;
    }
    int minute() {
        updateLocal();
        return local.tm_min;
    }
    int second() {
        updateLocal();
        return local.tm_sec;
    }
};
#endif // CPPTIME_H ///:~

```

标准C库函数对于时间有多种表示，它们都是类**Time**的一部分。但全部更新它们是没有必要的，所以**time\_t**被用作基本的表示法，**tm local**和ASCII字符表示法**asciiRep**都有一个标记来显示它们是否已被更新为当前的时间**time\_t**。两个私有函数**updateLocal()**和**updateAscii()**检查标记，并有条件地执行更新操作。

构造函数调用**mark()**函数时（用户也可以调用它，强迫对象表示当前时间）也就清除了两个标记，这时当地时间和ASCII表示法是无效的。函数**ascii()**调用**updateAscii()**，因为函数**ascii()**使用静态数据，假如它被调用，则这个静态数据被重写，所以**updateAscii()**把标准C库函数的结果拷贝到局部缓冲器里。函数**ascii()**返回值就是内部缓冲器的地址。

所有以**daylightSavings()**开始的函数都使用函数**updateLocal()**，这就使得复合的内联函数变得相当大。这似乎不划算，尤其是考虑到可能不经常调用这些函数。但这并不意味着所有的函数都应该用非内联函数。如果想让其他一些函数成为非内联函数的话，也至少让**updateLocal()**为内联函数，这样它的代码将被复制在所有的非内联函数里，也能消除函数调用时额外的开销。

下面是一个小的测试程序：

```

//: C09:Cpptime.cpp
// Testing a simple time class
#include "Cpptime.h"
#include <iostream>
using namespace std;
int main() {
    Time start;
    for(int i = 1; i < 1000; i++) {
        cout << i << ' ';
        if(i%10 == 0) cout << endl;
    }
    Time end;
    cout << endl;
    cout << "start = " << start.ascii();
    cout << "end = " << end.ascii();
    cout << "delta = " << end.delta(&start);
} ///:~

```

在这个例子里，创建了一个**Time**对象，然后执行一些时延动作，接着创建第2个**Time**对象来标记结束时间。这些用于显示开始时间、结束时间和消耗的时间。

### 9.3 带内联函数的Stash和Stack

引入了内联函数，现在，可以把**Stash**和**Stack**类变得更有效。

```

//: C09:Stash4.h
// Inline functions
#ifndef STASH4_H
#define STASH4_H
#include "../require.h"

class Stash {
    int size;           // Size of each space
    int quantity;       // Number of storage spaces
    int next;           // Next empty space
    // Dynamically allocated array of bytes:
    unsigned char* storage;
    void inflate(int increase);
public:
    Stash(int sz) : size(sz), quantity(0),
        next(0), storage(0) {}
    Stash(int sz, int initQuantity) : size(sz),
        quantity(0), next(0), storage(0) {
        inflate(initQuantity);
    }
    Stash::~Stash() {
        if(storage != 0)
            delete []storage;
    }
    int add(void* element);
    void* fetch(int index) const {
        require(0 <= index, "Stash::fetch (-)index");
        if(index >= next)
            return 0; // To indicate the end
        // Produce pointer to desired element:
        return &(storage[index * size]);
    }
};

```



```

    }
    int count() const { return next; }
};
#endif // STASH4_H ///:~

```

很明显，小函数作为内联函数工作是理想的，但要注意：两个最大的函数仍旧保留为非内联函数，因为要是把它们作为内联使用的话，很可能在性能上得不到什么改善。

```

//: C09:Stash4.cpp {0}
#include "Stash4.h"
#include <iostream>
#include <cassert>
using namespace std;
const int increment = 100;

int Stash::add(void* element) {
    if(next >= quantity) // Enough space left?
        inflate(increment);
    // Copy element into storage,
    // starting at next empty space:
    int startBytes = next * size;
    unsigned char* e = (unsigned char*)element;
    for(int i = 0; i < size; i++)
        storage[startBytes + i] = e[i];
    next++;
    return(next - 1); // Index number
}

void Stash::inflate(int increase) {
    assert(increase >= 0);
    if(increase == 0) return;
    int newQuantity = quantity + increase;
    int newBytes = newQuantity * size;
    int oldBytes = quantity * size;
    unsigned char* b = new unsigned char[newBytes];
    for(int i = 0; i < oldBytes; i++)
        b[i] = storage[i]; // Copy old to new
    delete [] (storage); // Release old storage
    storage = b; // Point to new memory
    quantity = newQuantity; // Adjust the size
} ///:~

```

测试程序再一次表明一切都正常运行。

```

//: C09:Stash4Test.cpp
//{L} Stash4
#include "Stash4.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
    Stash intStash(sizeof(int));
    for(int i = 0; i < 100; i++)
        intStash.add(&i);
}

```



```

for(int j = 0; j < intStash.count(); j++)
    cout << "intStash.fetch(" << j << ") = "
        << *(int*)intStash.fetch(j)
        << endl;
const int bufsize = 80;
Stash stringStash(sizeof(char) * bufsize, 100);
ifstream in("Stash4Test.cpp");
assure(in, "Stash4Test.cpp");
string line;
while(getline(in, line))
    stringStash.add((char*)line.c_str());
int k = 0;
char* cp;
while((cp = (char*)stringStash.fetch(k++))!=0)
    cout << "stringStash.fetch(" << k << ") = "
        << cp << endl;
} ///:~

```

这个程序同上面的测试程序相同，所以输出结果也基本一样。

**Stack**类更好地使用了内联函数。

```

//: C09:Stack4.h
// With inlines
#ifndef STACK4_H
#define STACK4_H
#include "../require.h"

class Stack {
    struct Link {
        void* data;
        Link* next;
        Link(void* dat, Link* nxt):
            data(dat), next(nxt) {}
    }* head;
public:
    Stack() : head(0) {}
    ~Stack() {
        require(head == 0, "Stack not empty");
    }
    void push(void* dat) {
        head = new Link(dat, head);
    }
    void* peek() const {
        return head ? head->data : 0;
    }
    void* pop() {
        if(head == 0) return 0;
        void* result = head->data;
        Link* oldHead = head;
        head = head->next;
        delete oldHead;
        return result;
    }
};
#endif // STACK4_H ///:~

```



注意：**Link**析构函数在前面的**Stack**版本中是以空的形式出现的，而在这里被删除了。在**pop()**中，表达式**delete oldHead**只是释放**Link**使用过的内存（它不销毁**Link**所指向的**data**对象）。

多数内联函数十分精细和明显，特别是对于**Link**尤其如此。甚至把**pop()**作为内联函数看起来也是合理的，尽管条件表达式或者局部变量对于使用内联函数的好处不明显。这里，函数很小，可以使用内联函数提高效率而无负面影响。

如果所有的函数都是内联函数，那么使用库就会变得相当简单，因为就像在上面的测试程序中所看到的一样，不需要进行库连接（注意并没有**Stack4.cpp**）。

```
//: C09:Stack4Test.cpp
//{T} Stack4Test.cpp
#include "Stack4.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char* argv[]) {
    requireArgs(argc, 1); // File name is argument
    ifstream in(argv[1]);
    assure(in, argv[1]);
    Stack textlines;
    string line;
    // Read file and store lines in the stack:
    while(getline(in, line))
        textlines.push(new string(line));
    // Pop the lines from the stack and print them:
    string* s;
    while((s = (string*)textlines.pop()) != 0) {
        cout << *s << endl;
        delete s;
    }
} ///:~
```

有时创建的类都是内联成员函数时，可以把整个类放在头文件中（我在本书中就跨越了这条界线），在程序开发的过程中，这是有益的，尽管编译时可能会花费更多的编译时间。一旦程序稍微稳定后，就可以返回去，在适当的地方把函数改为非成员函数。

## 9.4 内联函数和编译器

为了理解内联何时有效，应该先理解当编译器遇到一个内联函数时将做什么。对于任何函数，编译器在它的符号表里放入函数类型（即包括名字和参数类型的函数原型及函数的返回类型）。另外，当编译器看到内联函数和对内联函数体的进行分析没有发现错误时，就将对应于函数体的代码也放入符号表。代码是以源程序形式存放还是以编译过的汇编指令形式存放取决于编译器。

当调用一个内联函数时，编译器首先确保调用正确，即所有的参数类型必须满足：要么与函数参数表中的参数类型一样，要么编译器能够将其转换为正确类型，并且返回值在目标表达式里应该是正确类型或可改变为正确类型。当然，编译器为任何类型函数都是这样做的，并且这是与预处理器显著的不同之处，因为预处理器不能检查类型和进行转换。

假如所有的函数类型信息符合调用的上下文的话，内联函数代码就会直接替换函数调用，这消除了调用的开销，也考虑了编译器的进一步优化。假如内联函数也是成员函数，对象的地址(**this**)就会被放入合适的地方，这个动作当然也是预处理器不能完成的。

#### 9.4.1 限制

有两种编译器不能执行内联的情况。在这些情况下，它就像对非内联函数一样，根据内联函数定义和为函数建立存储空间，简单地将其转换为函数的普通形式。假如它必须在多重编译单元里做这些（通常将产生一个多定义错误），连接器就会被告知忽略多重定义。

假如函数太复杂，编译器将不能执行内联。这取决于特定的编译器，但对于大多数编译器这时都会放弃内联方式，这时内联将可能不能提高任何效率。一般地，任何种类的循环都被认为太复杂而不扩展为内联函数。循环在函数里可能比调用要花费更多的时间。假如函数仅由简单语句组成，编译器可能没有任何内联的麻烦，但假如函数有许多语句，调用函数的开销将比执行函数体的开销少多了。记住，每次调用一个大的内联函数，整个函数体就被插入在函数调用的地方，所以很容易使代码膨胀，而程序性能上没有任何显著的改进。（在本书中的一些例子中使用的内联函数可能超过一定合理的内联尺寸。）

假如要显式地或隐式地取函数地址，编译器也不能执行内联。因为这时编译器必须为函数代码分配内存从而产生一个函数的地址。但当地址不需要时，编译器仍将可能内联代码。

内联仅是编译器的一个建议，编译器不会被强迫内联任何代码。一个好的编译器将会内联小的、简单的函数，同时明智地忽略那些太复杂的内联。这将给我们想要的结果——具有宏效率的函数调用的真正的语义学。

#### 9.4.2 向前引用

如果猜想编译器执行内联函数时将会做什么事情，就可能会糊涂地认为限制比实际存在的要多。特别当一个内联函数在类中向前引用一个还没有声明的函数时，看起来好像编译器不能处理。

```
//: C09:EvaluationOrder.cpp
// Inline evaluation order

class Forward {
    int i;
public:
    Forward() : i(0) {}
    // Call to undeclared function:
    int f() const { return g() + 1; }
    int g() const { return i; }
};

int main() {
    Forward frwd;
    frwd.f();
} ///:~
```

函数f()调用g()，但此时还没有声明g()。这也能正常工作，因为C++语言规定：只有在类声明结束后，其中的内联函数才会被计算。



当然，如果`g()`反过来调用`f()`，就会产生递归调用，这对于编译器来说太复杂而不能执行内联。（应该在`f()`和`g()`中做一些测试，使其中一个有界可以退出，否则，递归将是无穷无尽的。）

### 9.4.3 在构造函数和析构函数里隐藏行为

在构造函数和析构函数中，可能易于认为内联的作用比它实际上更有效。构造函数和析构函数都可能隐藏行为，因为类可以包含子对象，子对象的构造函数和析构函数必须被调用。这些子对象可能是成员对象，或可能由于继承（继承将在第14章中介绍）而存在。下面是一个带成员对象的例子。

```
//: C09:Hidden.cpp
// Hidden activities in inlines
#include <iostream>
using namespace std;

class Member {
    int i, j, k;
public:
    Member(int x = 0) : i(x), j(x), k(x) {}
    ~Member() { cout << "~Member" << endl; }
};

class WithMembers {
    Member q, r, s; // Have constructors
    int i;
public:
    WithMembers(int ii) : i(ii) {} // Trivial?
    ~WithMembers() {
        cout << "~WithMembers" << endl;
    }
};

int main() {
    WithMembers wm(1);
} ///:~
```

**Member**的构造函数对于内联是足够简单的，它不做什么特别的事情。没有继承和成员对象会引起额外隐藏行为。但是在类**WithMembers**里，内联的构造函数和析构函数看起来似乎很直接和简单，但其实很复杂。成员对象`q`、`r`和`s`的构造函数和析构函数将被自动调用，这些构造函数和析构函数也是内联的，所以它们和普通的成员函数的差别是非常显著的。这并不是意味着应该使构造函数和析构函数定义为非内联的，只是在一些特定的情况下，这样做才是合理的。一般说来，快速地写代码来建立一个程序的初始“轮廓”时，使用内联函数经常是便利的。但假如要考虑效率，内联是值得注意的一个问题。

## 9.5 减少混乱

在本书里，把类里的内联定义做得简单和精练是非常有用的，因为这样更容易放在一页或一屏里，看起来更方便一些。但Dan Saks<sup>①</sup>指出，在一个真正的工程里，这将造成类接口

① 和Tom Plum 合著了《C++ Programming Guidelines》，Plum Hall, 1991.

混乱，因此使类难以使用。他用拉丁文*in situ* (在适当的位置上)来表示定义在类里的成员函数，并主张所有的定义都放在类外面以保持接口清楚。他认为这并不妨碍最优化。假如想优化，那么使用关键字**inline**。使用这个方法，前面（8.2.2节）的例子**Rectangle.cpp**修改如下：

```
//: C09:Noinsitu.cpp
// Removing in situ functions

class Rectangle {
    int width, height;
public:
    Rectangle(int w = 0, int h = 0);
    int getWidth() const;
    void setWidth(int w);
    int getHeight() const;
    void setHeight(int h);
};

inline Rectangle::Rectangle(int w, int h)
    : width(w), height(h) {}

inline int Rectangle::getWidth() const {
    return width;
}

inline void Rectangle::setWidth(int w) {
    width = w;
}

inline int Rectangle::getHeight() const {
    return height;
}

inline void Rectangle::setHeight(int h) {
    height = h;
}

int main() {
    Rectangle r(19, 47);
    // Transpose width & height:
    int iHeight = r.getHeight();
    r.setHeight(r.getWidth());
    r.setWidth(iHeight);
} ///:~
```

现在假如想比较一下内联函数与非内联函数的使用效果，可以简单地去掉关键字**inline**。（内联函数通常应该放在头文件里，但非内联函数必须放在它们自己的编译单元里。）假如想把函数放入文件，只用简单的剪切和粘贴操作就可完成。*in situ*函数需要更多的操作，且可能隐藏更多错误。这个方法的另外一个争论是可能总是对于函数定义使用一致的格式化类型，但有些并没有总是以*in situ*函数形式出现。

## 9.6 预处理器的更多特征

前面说过，我们几乎总是希望使用内联函数代替预处理器宏。然而当需要在标准C预处理器（通过继承也是C++预处理器）里使用3个特殊特征时却是例外：字符串定义、字符串拼接

和标志粘贴。字符串定义在本书的前面已作了介绍，字符串定义的完成是用#指示，它容许取一个标识符并把它转化为字符数组，然而字符串拼接在当两个相邻的字符串没有分隔符时发生，在这种情况下字符串组合在一起。在写调试代码时，这两个特征特别有用。

```
#define DEBUG(x) cout << #x " = " << x << endl
```

上面的这个定义可以打印任何变量的值。也可以得到一个跟踪信息，在此信息里打印出它们执行的语句。

```
#define TRACE(s) cerr << #s << endl; s
```

#s将输出语句字符。第2个s重申了该语句，所以这个语句被执行。当然，这可能会产生问题，尤其是在一行for循环中。

```
for(int i = 0; i < 100; i++)
    TRACE(f(i));
```

因为在TRACE()宏里实际上有两个语句，所以一行for循环只执行第一个。解决办法是在宏中用逗号代替分号。

### 9.6.1 标志粘贴

标志粘贴直接用“##”实现，在写代码时是非常有用的。它允许设两个标识符并把它们粘贴在一起自动产生一个新的标识符。例如：

```
#define FIELD(a) char* a##_string; int a##_size
class Record {
    FIELD(one);
    FIELD(two);
    FIELD(three);
    // ...
};
```

每次调用FIELD()宏，将产生一个保存字符数组的标识符和另一个保存字符数组长度的标识符。它不仅易读而且消除了编码出错，使维护更容易。

## 9.7 改进的错误检查

到目前为止，没有定义require.h中的函数却使用了它们（尽管assert()也被用在适当的地方来检查程序错误），现在该定义这个头文件了。在这里使用内联函数是便利的，因为它们允许放在头文件中，这样简化了包的使用过程。只要包含头文件，就不必担心连接一个实现文件。

应该注意异常处理机制（在本书的第2卷有详细的描述）为处理各种错误提供了一种更加有效的方法（特别是对于那些想恢复的错误），而不只是中止程序的运行。异常出现在诸如用户没有为一个文件提供足够的命令行参数，或者文件不能打开时。这时，程序不会继续运行。因此，可以调用标准的C库函数exit()。

下面的头文件将放在本书的根目录中，所以它可以从所有的章节里访问。

```
//: :require.h
// Test for error conditions in programs
// Local "using namespace std" for old compilers
```

```

#ifndef REQUIRE_H
#define REQUIRE_H
#include <cstdio>
#include <cstdlib>
#include <fstream>
#include <string>

inline void require(bool requirement,
    const std::string& msg = "Requirement failed"){
    using namespace std;
    if (!requirement) {
        fputs(msg.c_str(), stderr);
        fputs("\n", stderr);
        exit(1);
    }
}

inline void requireArgs(int argc, int args,
    const std::string& msg =
        "Must use %d arguments") {
    using namespace std;
    if (argc != args + 1) {
        fprintf(stderr, msg.c_str(), args);
        fputs("\n", stderr);
        exit(1);
    }
}

inline void requireMinArgs(int argc, int minArgs,
    const std::string& msg =
        "Must use at least %d arguments") {
    using namespace std;
    if (argc < minArgs + 1) {
        fprintf(stderr, msg.c_str(), minArgs);
        fputs("\n", stderr);
        exit(1);
    }
}

inline void assure(std::ifstream& in,
    const std::string& filename = "") {
    using namespace std;
    if (!in) {
        fprintf(stderr, "Could not open file %s\n",
            filename.c_str());
        exit(1);
    }
}

inline void assure(std::ofstream& out,
    const std::string& filename = "") {
    using namespace std;
    if (!out) {
        fprintf(stderr, "Could not open file %s\n",
            filename.c_str());
        exit(1);
    }
}

```



```

}
#endif // REQUIRE_H ///:~

```

默认值提供合理信息，必要时可以改变。

从上面可以看到，没有使用**char\***类型的参数，而是使用了**const string&**参数。这允许把**char\***和**string**作为这些函数的参数，一般说来，这样做更有用（在我们自己编码时也可能想这样）。

在**requireArgs()**和**requireMinArgs()**的定义中，增加了一个表示命令行中参数数目的参数，因为**argc**包括了总是作为第一个参数的程序名，所以**argc**比实际的命令行参数数目多一。

请注意在每一个函数中局部声明“**using namespace std**”的使用。这是因为声明不对时，编译器不会包含**namespace std**中标准的C库函数。这样将不能使用**namespace std**中的函数而导致编译错误。局部声明允许**require.h**同正确的和不正确的库一起工作，它不会为包含了这个头文件的任何人打开**namespace std**。

下面是一个测试**require.h**的简单程序。

```

//: C09:ErrTest.cpp
//{T} ErrTest.cpp
// Testing require.h
#include "../require.h"
#include <fstream>
using namespace std;

int main(int argc, char* argv[]) {
    int i = 1;
    require(i, "value must be nonzero");
    requireArgs(argc, 1);
    requireMinArgs(argc, 1);
    ifstream in(argv[1]);
    assure(in, argv[1]); // Use the file name
    ifstream nofile("nofile.xxx");
    // Fails:
    //! assure(nofile); // The default argument
    ofstream out("tmp.txt");
    assure(out);
} ///:~

```

为了打开文件也许想进一步地在**require.h**中加一个宏。

```

#define IFOPEN(VAR, NAME) \
    ifstream VAR(NAME); \
    assure(VAR, NAME);

```

可以像如下使用：

```
IFOPEN(in, argv[1])
```

刚开始，这种做法看起来是吸引人的，因为只要敲很少的代码。它虽然有一定的安全性，但最好还是避免这样做。应该注意：宏看起来像函数，但其行为方式不一样。它实际上创建一个对象（**in**），该对象的作用范围不仅仅在宏内。我们现在可以理解这一点，但是对于程序设计的新手和代码维护人员来说，令他们感到迷惑的不止这一点。所以，只要有可能就尽量不去使用预编译宏。

## 9.8 小结

能够隐藏类的底层实现是关键的，因为在以后有可能想修改这一实现。我们可能为了效率这样做，或为了对问题有更好的理解，或因为有些新类变得可用而想在实现里使用这些新类。任何危害实现隐蔽性的东西都会减少语言的灵活性。这样，内联函数就显得非常重要，因为它实际上消除了预处理器宏和伴随的问题。通过用内联函数方式，成员函数可以和预处理器宏一样有效。

当然，内联函数也许会在类定义里被多次使用。因为它更简单，所以程序设计者都会这样做。但这不是什么大问题，因为以后期待程序规模减少时，可以将函数移出内联而不影响它们的功能。程序开发的原则应该是“首先是使它可以工作，然后优化。”

## 9.9 练习

部分练习题的答案可以在本书的电子文档“*Annotated Solution Guide for Thinking in C++*”中找到，只需支付很少的费用就可以从<http://www.BruceEckel.com>得到这个电子文档。

- 9-1 写一个使用本章开头出现的**F()**宏的程序，证明它就像本章中所说的那样不能进行正确地扩展，修改宏并使程序能正确运行。
- 9-2 写一个使用本章开头出现的**FLOOR()**宏的程序，说明它在什么情况下不能正常运行。
- 9-3 修改**MacroSideEffects.cpp**，使**BAND()**能够正常运行。
- 9-4 创建两个功能相同的函数**f1()**和**f2()**，**f1()**是内联函数，**f2()**是非内联函数。使用**<ctime>**中的标准C库函数**clock()**标记这两个函数的开始点和结束点，比较它们看哪一个运行得更快，为了得到有效的数字，也许需要在计时循环中重复调用这两个函数。
- 9-5 对练习4中的函数代码的复杂性和大小作一下试验，看看对于内联函数和非内联函数在时间的消耗上，能否找到一个平衡点。如果可能，再在不同的编译器上试一试，并注意它们之间的差异。
- 9-6 证明内联函数默认为内部连接。
- 9-7 创建一个类，它包含一个整型数组。增加一个内联构造函数和一个内联成员函数**print()**。内联构造函数使用标准的C库函数**memset()**初始化对应于构造函数的参数（默认时为零）的数组，内联成员函数**print()**打印数组所有元素值。
- 9-8 把第5章中的例子**NestFriend.cpp**中的所有成员函数改成内联函数，并使它们为非*in situ*内联函数，也对于构造函数改造**initalize()**函数。
- 9-9 使用内联函数修改第8章中的**StringStack.cpp**。
- 9-10 创建一个称为**Hue**的**enum**，它包含**red**、**blue**和**yellow**。创建一个**color**类，该类包含一个**Hue**类型的数据成员，其构造函数用参数设置这个数据成员的值。增加访问函数用来获取和设置**Hue**这个数据成员的值，注意所有的函数都使用内联函数。
- 9-11 使用访问器和修改器的方法修改练习10中的程序。
- 9-12 修改程序**Cpptime.cpp**，使它从程序开始运行时开始计时，直到用户按确认(Enter)键或者回车键(Return)。
- 9-13 创建一个类，它带有两个内联成员函数，在类中定义的第一个成员函数调用第二个成员函数，而不需要提前声明。写一个主函数创建类的对象并调用第一个成员函数。

- 9-14 创建一个类**A**，它带有一个能声明自己的内联的默认的构造函数，再创建一个新类**B**，将**A**的一个对象作为**B**的成员，**B**的构造函数也是内联的，创建一个**B**类的对象数组，执行程序看看会出现什么情况。
- 9-15 从以前的练习的类中创建大量的对象并使用**Time**类来计算非内联构造函数和内联构造函数之间的时间差别（假如有剖析器(profiler)，也试着使用它。）
- 9-16 写一个带有一个**string**命令行参数的程序，写一个**for**循环，循环每执行一步就去掉**string**的一个字母并使用本章的**DEBUG()**宏打印**string**。
- 9-17 正确地修改**TRACE()**宏，使它成为本章所指定的特定宏，并使它能正确运行。
- 9-18 修改**FIELD()**宏，使它含有一个索引(**index**)号，创建一个类，它的成员由一些对**FIELD()**宏的调用组成，增加一个成员函数，它允许使用索引号查看域，写一个主函数**main()**测试这个类。
- 9-19 修改**FIELD()**宏，使它自动产生对每一个域访问的访问函数（数据应该仍旧是私有的）。创建一个类，它的成员由一些对**FIELD()**宏的调用组成，写一个主函数**main()**测试这个类。
- 9-20 写一个程序，它带两个命令行参数：第一个参数是一个整数，第二个参数是一个文件名，使用**require.h**以确保参数数目正确，并且整数在5到10之间，文件能够被成功地打开。
- 9-21 写一个使用**IFOPEN()**宏的程序，用它来打开一个文件并作为一个输入流，注意**ifstream**对象的创建以及它的作用域。
- 9-22 （高级）看看你的编译器怎样产生汇编代码。创建一个文件，它包含一个很小的函数和**main()**函数，**main()**调用这个小函数，分别产生这个小函数是内联和非内联时的汇编代码，证明内联版本比非内联版本的函数调用的开销要小。



## 名字控制

创建名字是程序设计过程中一项最基本的活动，当一个项目很大时，它会不可避免地包含大量的名字。

C++允许我们对名字的产生和名字的可见性进行控制，包括这些名字的存储位置以及名字的连接。

**static**这个关键字早在人们知道“重载”这个词的含义之前就在C语言中被重载了，并且在C++中又增加了另外的含义。关于**static**的所有使用最基本的概念是指“位置不变的某个东西”（如“静电”），不管这里是指在内存中的物理位置还是指在文件中的可见性。

在本章里，我们将看到**static**如何控制存储和可见性，还将看到一种通过C++的名字空间特征来控制访问名字的改进方法。我们还将发现怎样使用已经采用C语言编写和编译过的函数。

### 10.1 来自C语言中的静态元素

在C和C++中，**static**都有两种基本的含义，并且这两种含义经常是互相冲突的：

- 1) 在固定的地址上进行存储分配，也就是说对象是在一个特殊的静态数据区（*static data area*）上创建的，而不是每次函数调用时在堆栈上产生的。这也是静态存储的概念。
- 2) 对一个特定的编译单位来说是局部的（就像在后面将要看到的，这在C++中局限于类的范围）。这样，**static**控制名字的可见性（*visibility*），所以这个名字在这个单元或类之外是不可见的。这也描述了连接的概念，它决定连接器将看到哪些名字。

本节将着重讨论**static**的这两个含义，这些都是从C中继承来的。

#### 10.1.1 函数内部的静态变量

通常，在函数体内定义一个局部变量时，编译器在每次函数调用时使堆栈的指针向下移一个适当的位置，为这些局部变量分配内存。如果这个变量有一个初始化表达式，那么每当程序运行到此处，初始化就被执行。

然而，有时想在两次函数调用之间保留一个变量的值，可以通过定义一个全局变量来实现，但这样一来，这个变量就不仅仅只受这个函数的控制。C和C++都允许在函数内部定义一个**static**对象，这个对象将存储在程序的静态数据区中，而不是在堆栈中。这个对象只在函数第一次调用时初始化一次，以后它将在两次函数调用之间保持它的值。比如，下面的函数每次调用时都返回一个字符串中的下一个字符。

```
//: C10:StaticVariablesInFunctions.cpp
#include "../require.h"
#include <iostream>
using namespace std;

char oneChar(const char* charArray = 0) {
    static const char* s;
```



```

    if(charArray) {
        s = charArray;
        return *s;
    }
    else
        require(s, "un-initialized s");
    if(*s == '\0')
        return 0;
    return *s++;
}

char* a = "abcdefghijklmnopqrstuvwxyz";

int main() {
    // oneChar(); // require() fails
    oneChar(a); // Initializes s to a
    char c;
    while((c = oneChar()) != 0)
        cout << c << endl;
} ///:~

```

**static char\* s**在每次**oneChar()**调用时保留它的值，因为它存放在程序的静态数据区而不是存储在函数的堆栈中。当用一个字符指针作参数(**char\***)调用**oneChar()**时，参数值被赋给**s**，然后返回字符串的第一个字符。以后每次调用**oneChar()**都不用带参数，函数将使用默认参数**charArray**的默认值**0**，函数就会继续用以前初始化的**s**值取字符，直到它到达字符串的结尾标志——空字符为止，到这时，字符指针就不会再增加了，这样，指针不会越过字符串的末尾。

但是，如果调用**oneChar()**时没有参数而且**s**以前也没有初始化，那会怎样呢？也许会在定义**s**时提供一个初始值：

```
static char* s = 0;
```

但如果没有为一个内建类型的静态变量提供一个初始值的话，编译器也会确保在程序开始时它被初始化为零（转化为适当的类型），所以在**oneChar()**中，函数第一次调用时**s**将被赋值为零，这样**if(!s)**后面的程序就会被执行。

上例中**s**的初始化是很简单的，其实对一个静态对象的初始化（与其他对象的初始化一样）可以是任意的常量表达式，常量表达式中可以出现常量及在此之前已声明过的变量和函数。

应该知道：上面的函数很容易产生多线程问题；无论什么时候设计一个包含静态变量的函数时，都应该记住多线程问题。

#### 10.1.1.1 函数内部的静态对象

关于一般的静态变量的规则同样适用于用户自定义的静态对象，而且它同样也必须有初始化操作。但是，零赋值只对内建类型有效，用户自定义类型必须用构造函数来初始化。因此，如果在定义一个静态对象时没有指定构造函数参数，这个类就必须有默认的构造函数。请看下例：

```

//: C10:StaticObjectsInFunctions.cpp
#include <iostream>
using namespace std;

class X {
    int i;

```

```

public:
    X(int ii = 0) : i(ii) {} // Default
    ~X() { cout << "X::~X()" << endl; }
};

void f() {
    static X x1(47);
    static X x2; // Default constructor required
}

int main() {
    f();
} ///:~

```

在函数`f()`内部定义一个静态的`X`类型的对象，它可以用带参数的构造函数来初始化，也可以用默认构造函数。程序控制第一次转到对象的定义点时，而且只有第一次时，才需要执行构造函数。

#### 10.1.1.2 静态对象的析构函数

静态对象的析构函数（包括静态存储的所有对象，不仅仅是上例中的局部静态对象）在程序从`main()`中退出时，或者标准的C库函数`exit()`被调用时才被调用。多数情况下`main()`函数的结尾也是调用`exit()`来结束程序的。这意味着在析构函数内部使用`exit()`是很危险的，因为这样导致了无穷的递归调用。但如果用标准的C库函数`abort()`来退出程序，静态对象的析构函数并不会被调用。

可以用标准C库函数`atexit()`来指定当程序跳出`main()`（或调用`exit()`）时应执行的操作。在这种情况下，在跳出`main()`或调用`exit()`之前，用`atexit()`注册的函数可以在所有对象的析构函数之前被调用。

同普通对象的销毁一样，静态对象的销毁也是按与初始化时相反的顺序进行的。当然只有那些已经被创建的对象才会被销毁。幸运的是，开发工具会记录对象初始化的顺序和那些已被创建的对象。全局对象总是在`main()`执行之前被创建，在退出`main()`时销毁。如果一个包含局部静态对象的函数从未被调用过，那么这个对象的构造函数也就不会执行，这样自然也不会执行析构函数。请看下例：

```

//: C10:StaticDestructors.cpp
// Static object destructors
#include <fstream>
using namespace std;
ofstream out("statdest.out"); // Trace file

class Obj {
    char c; // Identifier
public:
    Obj(char cc) : c(cc) {
        out << "Obj::Obj() for " << c << endl;
    }
    ~Obj() {
        out << "Obj::~Obj() for " << c << endl;
    }
};

Obj a('a'); // Global (static storage)

```



```
// Constructor & destructor always called

void f() {
    static Obj b('b');
}

void g() {
    static Obj c('c');
}

int main() {
    out << "inside main()" << endl;
    f(); // Calls static constructor for b
    // g() not called
    out << "leaving main()" << endl;
} ///:~
```

在**Obj**中，**char c**的作用就像一个标识符，构造函数和析构函数就可以通过**c**显示出当前正在操作的对象信息。而**Obj a**是一个全局的**Obj**类的对象，所以构造函数总是在**main()**函数之前就被调用。但函数**f()**内的**Obj**类的静态对象**b**和函数**g()**内的静态对象**c**的构造函数只在这些函数被调用时才起作用。

为了说明哪些构造函数与析构函数被调用，在**main()**中只调用了**f()**，程序的输出结果为：

```
Obj::Obj() for a
inside main()
Obj::Obj() for b
leaving main()
Obj::~Obj() for b
Obj::~Obj() for a
```

在执行**main()**函数之前，对象**a**的构造函数即被调用，而**b**的构造函数只是因为**f()**的调用而调用。当退出**main()**函数时，所有被创建的对象析构函数按创建时相反的顺序被调用。这意味着如果**g()**被调用，对象**b**和**c**的析构函数的调用顺序依赖于**g()**和**f()**的调用顺序。

注意跟踪文件**ofstream**的对象**out**也是一个静态对象，因为它定义在所有函数之外，位于静态存储区。它的定义（因为不用**extern**定义）应该出现在文件的一开始，在**out**的任何可能的使用出现之前，这一点很重要，否则就可能在一个对象初始化之前使用它。

在C++中，全局静态对象的构造函数是在**main()**之前调用的，所以现在有了一个在进入**main()**之前执行一段代码的简单的、可移植的方法，并且可以在退出**main()**之后用析构函数执行代码。在C中要做到这一点，就显得很繁琐，我们将不得不熟悉编译器开发商的汇编语言的开始代码。

### 10.1.2 控制连接

一般情况下，在文件作用域（*file scope*）内的所有名字（即不嵌套在类或函数中的名字）对程序中的所有翻译单元来说都是可见的。这就是所谓的外部连接（*external linkage*），因为在连接时这个名字对连接器来说是可见的，对单独的翻译单元来说，它是外部的。全局变量和普通函数都有外部连接。

有时可能想限制一个名字的可见性。想让一个变量在文件范围内是可见的，这样这个文件中的所有函数都可以使用它，但不想让这个文件之外的函数看到或访问该变量，或不想这

个变量的名字与外部的标识符相冲突。

在文件作用域内，一个被明确声明为**static**的对象或函数的名字对翻译单元（用本书的术语来说也就是出现声明的**.cpp**文件）来说是局部于该单元的。这些名字有内部连接（*internal linkage*）。这意味着可以在其他的翻译单元中使用同样的名字，而不会发生名字冲突。

内部连接的一个好处是这个名字可以放在一个头文件中而不用担心连接时发生冲突。那些通常放在头文件里的名字，如常量、内联函数，在默认情况下都是内部连接的（当然常量只有在C++中默认情况下是内部连接的，在C中它默认为外部连接）。注意连接只引用那些在连接/装载期间有地址的成员，因此类声明和局部变量并不连接。

#### 10.1.2.1 冲突问题

下面例子说明了**static**的两个含义是怎样彼此交叉的。所有的全局对象都是隐含为静态存储的，所以如果定义（在文件作用域）

```
int a = 0;
```

则**a**被存储在程序的静态数据区，在进入**main()**函数之前，**a**即已初始化了。另外，**a**对所有的翻译单元都是全局可见的。用可见性术语来讲，**static**（只在翻译单元内可见）的反义是**extern**，它明确地声明了这个名字对所有的翻译单元都是可见的。所以上面的定义和下面的定义是相同的。

```
extern int a = 0;
```

但如果这样定义：

```
static int a = 0;
```

只不过改变了**a**的可见性，现在**a**成了一个内部连接，但存储类型没有改变——对象总是驻留在静态数据区，而不管是**static**还是**extern**。

一旦进入局部变量，**static**就不会再改变变量的可见性（这时**extern**是没有意义的），而只是改变变量的存储类型。

如果把局部变量声明为**extern**，这意味着某处已经存在一个存储区（所以该变量对函数来说实际上是全局的），请看下面的例子。

```
//: C10:LocalExtern.cpp
//{L} LocalExtern2
#include <iostream>

int main() {
    extern int i;
    std::cout << i;
} ///:~

//: C10:LocalExtern2.cpp {O}
int i = 5;
///:~
```

对函数名（非成员函数），**static**和**extern**只会改变它们的可见性，所以如果说：

```
extern void f();
```

它和没有修饰时的声明是一样的：

```
void f();
```

如果定义：

```
static void f();
```

它意味着f()只在本翻译单元内是可见的，这有时称作文件静态 (file static)。

### 10.1.3 其他存储类型说明符

我们会看到**static**和**extern**用得很普遍。另外还有用得较少的两个存储类型说明符。一个是**auto**，人们几乎不用它，因为它告诉编译器这是一个局部变量。**auto**是“automatic”的缩写，它指明编译器自动为该变量分配存储空间的方法。实际上编译器总是可以从变量定义时的上下文中判断出这是一个局部变量，所以**auto**是多余的。

还有一个是**register**，它说明的也是局部 (**auto**) 变量，但它告诉编译器这个特殊的变量要经常用到，所以编译器应该尽可能地让它保存在寄存器中。它用于优化代码。但各种编译器对这种类型的变量处理方式也不尽相同，它们有时会忽略这种存储类型的指定。一般，如果要用到这个变量的地址，**register**指定符通常都会被忽略。应该避免用**register**类型，因为编译器在优化代码方面通常比我们做得更好。

## 10.2 名字空间

虽然名字可以嵌套在类中，但全局函数、全局变量以及类的名字还是在同一个全局名字空间中。虽然**static**关键字可以使变量和函数实行内部连接 (使它们文件静态)，从而做到一定的控制。但在一个大项目中，如果对全局的名字空间缺乏控制就会引起很多问题。为了解决这些问题，开发商常常使用冗长、难懂的名字，以使冲突减少，但这样我们不得不一个一个地敲这些名字 (**typedef**常常用来简化这些名字)。但这不是一个很好的解决方法。

可以用C++的名字空间 (namespace) 特征，把一个全局名字空间分成多个可管理的小空间。关键字**namespace**，如同**class**、**struct**、**enum**和**union**一样，把它们的成员的名字放到了不同的空间中去，尽管其他的关键字有其他的目的，但**namespace**惟一的目的是产生一个新的名字空间。

### 10.2.1 创建一个名字空间

创建一个名字空间与创建一个类非常相似：

```
//: C10:MyLib.cpp
namespace MyLib {
    // Declarations
}
int main() {} ///:~
```

这就产生了一个新的名字空间，其中包含了各种声明。然而，**namespace**与**class**、**struct**、**union**和**enum**有着明显的区别：

- **namespace**只能在全局范围内定义，但它们之间可以互相嵌套。
- 在**namespace**定义的结尾，右花括号的后面不必跟一个分号。
- 可以按类的语法来定义一个**namespace**，定义的内容可在多个头文件中延续，就好像重复定义这个**namespace**一样。

```
//: C10:Header1.h
#ifndef HEADER1_H
```

```

#define HEADER1_H
namespace MyLib {
    extern int x;
    void f();
    // ...
}

#endif // HEADER1_H ///:~
//: C10:Header2.h
#ifndef HEADER2_H
#define HEADER2_H
#include "Header1.h"
// Add more names to MyLib
namespace MyLib { // NOT a redefinition!
    extern int y;
    void g();
    // ...
}

#endif // HEADER2_H ///:~
//: C10:Continuation.cpp
#include "Header2.h"
int main() {} ///:~

```

- 一个**namespace**的名字可以用另一个名字来作它的别名，这样就不必敲打那些开发商提供的冗长的名字了。

```

//: C10:BobsSuperDuperLibrary.cpp
namespace BobsSuperDuperLibrary {
    class Widget { /* ... */ };
    class Poppit { /* ... */ };
    // ...
}
// Too much to type! I'll alias it:
namespace Bob = BobsSuperDuperLibrary;
int main() {} ///:~

```

- 不能像类那样去创建一个名字空间的实例。

#### 10.2.1.1 未命名的名字空间

每个翻译单元都可包含一个未命名的名字空间——可以不用标识符而只用“**namespace**”增加一个名字空间。

```

//: C10:UnnamedNamespaces.cpp
namespace {
    class Arm { /* ... */ };
    class Leg { /* ... */ };
    class Head { /* ... */ };
    class Robot {
        Arm arm[4];
        Leg leg[16];
        Head head[3];
        // ...
    } xanthan;
    int i, j, k;
}
int main() {} ///:~

```



在这个空间中的名字自动地在翻译单元内无限制地有效。但要确保每个翻译单元只有一个未命名的名字空间。如果把一个局部名字放在一个未命名的名字空间中，不需要加上**static**说明就可以让它们作内部连接。

#### 10.2.1.2 友元

可以在一个名字空间的类定义之内插入 (*inject*) 一个友元 (**friend**) 声明:

```
//: C10:FriendInjection.cpp
namespace Me {
    class Us {
        //...
        friend void you();
    };
}
int main() {} ///:~
```

这样函数**you()**就成了名字空间**Me**的一个成员。

### 10.2.2 使用名字空间

在一个名字空间中引用一个名字可以采取两种方法: 第一种方法是用作用域运算符, 第二种方法是用**using**指令把所有名字引入到名字空间中。

#### 10.2.2.1 作用域解析

名字空间中的任何名字都可以用作用域运算符作明确地指定, 就像引用一个类中的名字一样:

```
//: C10:ScopeResolution.cpp
namespace X {
    class Y {
        static int i;
    public:
        void f();
    };
    class Z;
    void func();
}
int X::Y::i = 9;
class X::Z {
    int u, v, w;
public:
    Z(int i);
    int g();
};
X::Z::Z(int i) { u = v = w = i; }
int X::Z::g() { return u = v = w = 0; }
void X::func() {
    X::Z a(1);
    a.g();
}
int main() {} ///:~
```

注意定义**X::Y::i**就像引用一个类**Y**的数据成员一样容易, **Y**如同被嵌套在类**X**中而不像是

被嵌套在名字空间`X`中。

到目前为止，名字空间看上去很像类。

#### 10.2.2.2 使用指令

用**using**关键字可以让我们立即进入整个名字空间，摆脱输入一个名字空间中完整标识符的烦恼。这种**using**和**namespace**关键字的搭配使用称为使用指令 (*using directive*)。using 关键字声明了一个名字空间中的所有名字是在当前范围内，所以可以很方便地使用这些未限定的名字。如果以一个简单的名字空间开始：

```
//: C10:NamespaceInt.h
#ifndef NAMESPACEINT_H
#define NAMESPACEINT_H
namespace Int {
    enum sign { positive, negative };
    class Integer {
        int i;
        sign s;
    public:
        Integer(int ii = 0)
            : i(ii),
              s(i >= 0 ? positive : negative)
        {}
        sign getSign() const { return s; }
        void setSign(sign sgn) { s = sgn; }
        // ...
    };
}
#endif // NAMESPACEINT_H ///:~
```

**using**指令的用途之一就是要把名字空间`Int`中的所有名字引入到另一个名字空间中，让这些名字嵌套在那个名字空间中。

```
//: C10:NamespaceMath.h
#ifndef NAMESPACEMATH_H
#define NAMESPACEMATH_H
#include "NamespaceInt.h"
namespace Math {
    using namespace Int;
    Integer a, b;
    Integer divide(Integer, Integer);
    // ...
}
#endif // NAMESPACEMATH_H ///:~
```

可以在一个函数中声明名字空间`Int`中的所有名字，但是让这些名字嵌套在这个函数中。

```
//: C10:Arithmetic.cpp
#include "NamespaceInt.h"
void arithmetic() {
    using namespace Int;
    Integer x;
    x.setSign(positive);
}
int main(){} ///:~
```





如果不用**using**指令，在这个名字空间的所有名字都需要被完全限定。

**using** 指令有一个缺点，那就是看起来不那么直观，引入名字的可见性的范围是在使用**using**的地方。可以不考虑使用**using** 指令的名字，就像它们已经被全局声明过，现在变为这个范围。

```
//: C10:NamespaceOverriding1.cpp
#include "NamespaceMath.h"
int main() {
    using namespace Math;
    Integer a; // Hides Math::a;
    a.setSign(negative);
    // Now scope resolution is necessary
    // to select Math::a :
    Math::a.setSign(positive);
} ///:~
```

如果有第二个名字空间，它包含了名字空间**Math**的某些名字：

```
//: C10:NamespaceOverriding2.h
#ifndef NAMESPACEOVERRIDING2_H
#define NAMESPACEOVERRIDING2_H
#include "NamespaceInt.h"
namespace Calculation {
    using namespace Int;
    Integer divide(Integer, Integer);
    // ...
}
#endif // NAMESPACEOVERRIDING2_H ///:~
```

因为这个名字空间也是用**using**指令来引入的，这样就可能产生冲突。不过，这种二义性出现在名字的使用时，而不是在**using**指令使用时。

```
//: C10:OverridingAmbiguity.cpp
#include "NamespaceMath.h"
#include "NamespaceOverriding2.h"
void s() {
    using namespace Math;
    using namespace Calculation;
    // Everything's ok until:
    //! divide(1, 2); // Ambiguity
}
int main() {} ///:~
```

这样，即使永远不产生歧义性，使用**using**指令引入带名字冲突的名字空间也是可能的。

### 10.2.2.3 使用声明

可以用使用声明 (*using declaration*) 一次性引入名字到当前范围内。这种方法不像**using**指令那样把那些名字当成当前范围的全局名来看待，**using**声明是在当前范围之内进行的一个声明，这就意味着在这个范围内它可以不顾来自**using**指令的名字。

```
//: C10:UsingDeclaration.h
#ifndef USINGDECLARATION_H
#define USINGDECLARATION_H
namespace U {
```

```

    inline void f() {}
    inline void g() {}
}
namespace V {
    inline void f() {}
    inline void g() {}
}
#endif // USINGDECLARATION_H ///:~

//: C10:UsingDeclaration1.cpp
#include "UsingDeclaration.h"
void h() {
    using namespace U; // Using directive
    using V::f; // Using declaration
    f(); // Calls V::f();
    U::f(); // Must fully qualify to call
}
int main() {} ///:~

```

**using**声明给出了标识符的完整的名字，但没有了类型方面的信息。也就是说，如果名字空间中包含了一组用相同名字重载的函数，**using**声明就声明了这个重载的集合内的所有函数。

可以把**using**声明放在任何一般的声明可以出现的地方。**using**声明与普通声明只有一点不同：**using**声明可以引起一个函数用相同的参数类型来重载（这在一般的重载中是不允许的）。当然这种不确定性要到使用时才表现出来，而不是在声明时。

**using**声明也可以出现在一个名字空间内，其作用与在其他地方时一样：

```

//: C10:UsingDeclaration2.cpp
#include "UsingDeclaration.h"
namespace Q {
    using U::f;
    using V::g;
    // ...
}
void m() {
    using namespace Q;
    f(); // Calls U::f();
    g(); // Calls V::g();
}
int main() {} ///:~

```

一个**using**声明是一个别名，它允许在不同的名字空间声明同样的函数。如果不想由于引入不同名字空间而导致重复定义一个函数时，可以使用**using**声明，它不会引起任何二义性和重复。

### 10.2.3 名字空间的使用

上面所介绍的一些规则刚开始时也许会使我们感到气馁，特别是当我们知道将来一直使用它们会有什么感觉时，尤其如此。一般说来，只要真正理解了它们的工作机理，使用它们也会变得非常简单。需要记住的关键问题是当引入一个全局**using**指令时（可以在任何范围之外通过使用**using namespace**），就已经为那个文件打开了该名字空间。对于一个实现文件（一个.cpp文件）来说，这通常是一个好方法，因为只有在该文件编译结束时，**using**指令才会起作

用。也就是说，它不会影响任何其他文件，所以可以每次在一个实现文件中调整对名字空间的控制。例如，如果发现由于在一个特定的实现文件中使用太多的**using**指令而产生名字冲突，就要对该文件做简单的改变，以致使用明确的限定或者**using**声明来消除名字冲突，这样不用修改其他的实现文件。

头文件的情况与此不同。不要把一个全局的**using**指令引入到一个头文件中，因为那将意味着包含这个头文件的任何其他头文件也会打开这个名字空间（头文件可以被另一个头文件包含）。

所以，在头文件中，最好使用明确的限定或者被限定在一定范围内的**using**指令和**using**声明。在本书中将讨论这种用法，通过这种方法，就不会“污染”全局名字空间和后退到C++的名字空间引入前的世界。

### 10.3 C++中的静态成员

有时需要为某个类的所有对象分配一个单一的存储空间。在C语言中，可以用全局变量，但这样很不安全。全局数据可以被任何人修改，而且，在一个大项目中，它很容易与其他的名字相冲突。如果可以把一个数据当成全局变量那样去存储，但又被隐藏在类的内部，并且清楚地与这个类相联系，这种处理方法当然是最理想的了。

这一点可以用类的静态数据成员来实现。类的静态成员拥有一块单独的存储区，而不管创建了多少个该类的对象。所有的这些对象的静态数据成员都共享这一块静态存储空间，这就为这些对象提供了一种互相通信的方法。但静态数据属于类，它的名字只在类的范围内有效，并且可以是**public**（公有的）、**private**（私有的）或者**protected**（保护的）。

#### 10.3.1 定义静态数据成员的存储

因为类的静态数据成员有着单一的存储空间而不管产生了多少个对象，所以存储空间必须在一个单独的地方定义。编译器不会分配存储空间。如果一个静态数据成员被声明但没有定义时，连接器会报告一个错误。

定义必须出现在类的外部（不允许内联）而且只能定义一次，因此它通常放在一个类的实现文件中。这种规定常常让人感到很麻烦，但它实际上是很合理的。例如，在一个类中定义一个静态数据成员如下：

```
class A {
    static int i;
public:
    //...
};
```

之后，必须在定义文件中为静态数据成员定义存储区：

```
int A::i = 1;
```

如果要定义了一个普通的全局变量，可以这样：

```
int i = 1;
```

在这里，类名和作用域运算符用于指定了**A::i**。

有些人对**A::i**是私有的这点感到疑惑不解，可是在这里似乎在公开地直接对它处理。这不

是破坏了类结构的保护性吗？有两个原因可以保证它绝对的安全。第一，这些变量的初始化惟一合法的地方是在定义时。事实上，如果静态数据成员是一个带构造函数的对象时，可以调用构造函数来代替“=”操作符；第二，一旦这些数据被定义了，最终的用户就不能再定义它——否则连接器会报告错误。而且这个类的创建者被迫产生这个定义，否则这些代码在测试时无法连接。这就保证了定义只出现一次并且它是由类的构造者来控制的。

静态成员的初始化表达式是在一个类的作用域内，请看下例：

```
//: C10:Statinit.cpp
// Scope of static initializer
#include <iostream>
using namespace std;

int x = 100;

class WithStatic {
    static int x;
    static int y;
public:
    void print() const {
        cout << "WithStatic::x = " << x << endl;
        cout << "WithStatic::y = " << y << endl;
    }
};

int WithStatic::x = 1;
int WithStatic::y = x + 1;
// WithStatic::x NOT ::x

int main() {
    WithStatic ws;
    ws.print();
} ///:~
```

这里，**withStatic::**限定符把**withStatic**的作用域扩展到全部定义中。

#### 10.3.1.1 静态数组的初始化

第8章介绍了静态常量（**static const**）变量，它允许在一个类体中定义一个常量值。也可以创建静态对象数组，包括**const**数组与**非const**数组。这同前面的语法是一致的。

```
//: C10:StaticArray.cpp
// Initializing static arrays in classes
class Values {
    // static consts are initialized in-place:
    static const int scSize = 100;
    static const long scLong = 100;
    // Automatic counting works with static arrays.
    // Arrays, Non-integral and non-const statics
    // must be initialized externally:
    static const int scInts[];
    static const long scLongs[];
    static const float scTable[];
    static const char scLetters[];
    static int size;
    static const float scFloat;
```



```

    static float table[];
    static char letters[];
};

int Values::size = 100;
const float Values::scFloat = 1.1;

const int Values::scInts[] = {
    99, 47, 33, 11, 7
};

const long Values::scLongs[] = {
    99, 47, 33, 11, 7
};

const float Values::scTable[] = {
    1.1, 2.2, 3.3, 4.4
};

const char Values::scLetters[] = {
    'a', 'b', 'c', 'd', 'e',
    'f', 'g', 'h', 'i', 'j'
};

float Values::table[4] = {
    1.1, 2.2, 3.3, 4.4
};

char Values::letters[10] = {
    'a', 'b', 'c', 'd', 'e',
    'f', 'g', 'h', 'i', 'j'
};

int main() { Values v; } ///:~

```

利用全部类型的静态常量，可以在类内提供这些定义，但是对于其他的对象（包括全部类型的数组，甚至它们为静态常量），必须为这些成员提供专门的外部定义。这些定义是内部连接的，所以可以把它放在头文件中，初始化静态数组的方法与其他聚合类型的初始化一样，包括自动计数。

也可以创建类的静态常量对象和这样的对象的数组。不过，不能使用“内联语法”初始化它们，这种语法对全部的内建类型的静态常量有效。

```

//: C10:StaticObjectArrays.cpp
// Static arrays of class objects
class X {
    int i;
public:
    X(int ii) : i(ii) {}
};

class Stat {
    // This doesn't work, although
    // you might want it to:
    //! static const X x(100);
    // Both const and non-const static class

```



```

    // objects must be initialized externally:
    static X x2;
    static X xTable2[];
    static const X x3;
    static const X xTable3[];
};

X Stat::x2(100);

X Stat::xTable2[] = {
    X(1), X(2), X(3), X(4)
};

const X Stat::x3(100);
const X Stat::xTable3[] = {
    X(1), X(2), X(3), X(4)
};

int main() { Stat v; } ///:~

```

类对象的常量和非常量静态数组的初始化必须以相同的方式执行，它们遵守典型的静态定义语法。

### 10.3.2 嵌套类和局部类

可以很容易地把一个静态数据成员放在另一个类的嵌套类中。这样的成员的定义显然是上节中情况的扩展——只须用另一种级别的作用域指定。然而不能在局部类（在函数内部定义的类）中有静态数据成员。因而，如下例：

```

//: C10:Local.cpp
// Static members & local classes
#include <iostream>
using namespace std;

// Nested class CAN have static data members:
class Outer {
    class Inner {
        static int i; // OK
    };
};

int Outer::Inner::i = 47;

// Local class cannot have static data members:
void f() {
    class Local {
    public:
    //! static int i; // Error
        // (How would you define i?)
    } x;
}

int main() { Outer x; f(); } ///:~

```



可以看到一个局部类中有与静态成员直接相关的问题。为了定义数据成员，怎样才能在文件范围描述它呢？实际上很少使用局部类。

### 10.3.3 静态成员函数

像静态数据成员一样，也可以创建一个静态成员函数，它为类的全体对象服务而不是为一个类的特殊对象服务。这样就不需要定义一个全局函数，减少了全局或局部名字空间的占用，把这个函数移到了类的内部。当产生一个静态成员函数时，也就表达了与一个特定类的联系。

可以用普通的方法调用静态成员函数，用点“.”和箭头“->”把它与一个对象相联系。然而，调用静态成员函数的一个更典型的方法是自我调用，这不需要任何具体的对象，而是像下面使用作用域运算符：

```
//: C10:SimpleStaticMemberFunction.cpp
class X {
public:
    static void f(){};
};

int main() {
    X::f();
} ///:~
```

当在一个类中看到静态成员函数时，要记住：类的设计者是想把这些函数与整个类在概念上关联起来。

静态成员函数不能访问一般的数据成员，而只能访问静态数据成员，也只能调用其他的静态成员函数。通常，当前对象的地址（**this**）是被隐式地传递到被调用的函数的。但一个静态成员函数没有**this**，所以它无法访问一般的成员。这样使用静态成员函数在速度上可以比全局函数有少许的增长，它不仅没有传递**this**所需的额外开销，而且还有使函数在类内的好处。

对于数据成员来说，**static**关键字指定它对类的所有对象来说，都只占有相同的一块存储空间。与定义对象的静态使用相对应，静态函数意味着对这个函数的所有调用来说，一个局部变量只有一份拷贝。

下面是一个静态数据成员和静态成员函数在一起使用的例子：

```
//: C10:StaticMemberFunctions.cpp
class X {
    int i;
    static int j;
public:
    X(int ii = 0) : i(ii) {
        // Non-static member function can access
        // static member function or data:
        j = i;
    }
    int val() const { return i; }
    static int incr() {
        //! i++; // Error: static member function
        // cannot access non-static member data
        return ++j;
    }
};
```



```

    }
    static int f() {
        //! val(); // Error: static member function
        // cannot access non-static member function
        return incr(); // OK -- calls static
    }
};

int X::j = 0;

int main() {
    X x;
    X* xp = &x;
    x.f();
    xp->f();
    X::f(); // Only works with static members
} ///:~

```

因为静态成员函数没有**this**指针，所以它既不能访问非静态的数据成员，也不能调用非静态的成员函数。

注意在**main()**中，一个静态成员可以用点或箭头来选取，把那个函数与一个对象联系起来，但也可以不与对象相联系（因为一个静态成员是与一个类相连，而不是与一个特定的对象相连），而是用类的名字和作用域运算符。

这里有一个有趣的特点：因为静态成员对象的初始化方法，所以可以把上述类的一个静态数据成员放到那个类的内部。下面是一个例子，它把构造函数变成私有的，这样**Egg**类只有一个惟一的对象存在，可以访问那个对象，但不能产生任何新的**Egg**对象。

```

//: C10:Singleton.cpp
// Static member of same type, ensures that
// only one object of this type exists.
// Also referred to as the "singleton" pattern.
#include <iostream>
using namespace std;

class Egg {
    static Egg e;
    int i;
    Egg(int ii) : i(ii) {}
    Egg(const Egg&); // Prevent copy-construction
public:
    static Egg* instance() { return &e; }
    int val() const { return i; }
};

Egg Egg::e(47);

int main() {
    //! Egg x(1); // Error -- can't create an Egg
    // You can access the single instance:
    cout << Egg::instance()->val() << endl;
} ///:~

```

**E**的初始化出现在类的声明完成后，所以编译器已有足够的信息为对象分配空间并调用构



构造函数。

为了完全防止创建其他对象，还需要再做如下工作：增加一个叫做拷贝构造函数（*copy constructor*）的私有构造函数。到目前为止，还不知道为什么必须这样做，因为在下章中才会讨论拷贝构造函数。然而，如果删除上面例子中定义的拷贝构造函数，那么就能像下面那样创建一个Egg对象。

```
Egg e = *Egg::instance();
Egg e2(*Egg::instance());
```

这两条语句都使用了拷贝构造函数，所以为了禁止这种可能性，拷贝构造函数声明为私有的（不需要定义，因为它不会被调用）。第11章的大部分内容是对拷贝构造函数的讨论，所以，通过第11章的学习后，我们会明白是怎么回事。

## 10.4 静态初始化的相依性

在一个指定的翻译单元中，静态对象的初始化顺序严格按照对象在该单元中定义出现的顺序。而清除的顺序则与初始化的顺序正好相反。

但是，对于作用域为多个翻译单元的静态对象来说，不能保证严格的初始化顺序，也没有办法来指定这种顺序。这可能会引起一些问题。下面的例子如果包含一个文件就会立即引起灾难（它会暂停一些简单的操作系统的运行，中止进程）。

```
// First file
#include <fstream>
std::ofstream out("out.txt");
```

另一个文件在它的初始表达式之一中用到了out对象：

```
// Second file
#include <fstream>
extern std::ofstream out;
class Oof {
public:
    Oof() { std::out << "ouch"; }
} oof;
```

这个程序可能运行，也可能不能运行。如果在建立可执行文件时第一个文件先初始化，那么就不会有问题，但如果第二个文件先初始化，Oof的构造函数依赖于out的存在，而此时out还没有创建，于是就会引起混乱。

这种情况只会在相互依赖的静态对象的初始化时出现。在一个翻译单元内的一个函数的第一次调用之前，但在进入main()之后，这个翻译单元内的静态对象都被初始化。如果静态对象位于不同的文件中，则不能确定这些静态对象的初始化顺序。

在ARM<sup>①</sup>中可以看到一个更微妙的例子，在一个文件中：

```
extern int y;
int x = y + 1;
```

在另一个文件中

```
extern int x;
```

① 《The Annotated C++ Reference Manual》,Bjarne Stroustrup和Margaret Ellis著，1990年，20~21页。

```
int y = x + 1;
```

对所有的静态对象，连接装载机制在程序员指定的动态初始化发生前保证一个静态成员初始化为零。在前一个例子中，**fstream out**对象的存储空间赋零并没有特别的意义，所以它在构造函数调用前确实是未定义的。然而，对内建数据类型，初始化为零是有意义的，所以如果文件按上面的顺序被初始化，**y**开始被初始化为零，所以**x**变成**1**，而后**y**被动态初始化为**2**。然而，如果初始化的顺序颠倒过来，**x**被静态初始化为零，**y**被初始化为**1**，而后**x**被初始化为**2**。

程序员必须意识到这些，因为他们可能会在编程时遇到互相依赖的静态变量的初始化问题，程序可能在一个平台上工作正常，当把它移到另一个编译环境时，突然莫名其妙地不工作了。

#### 10.4.1 怎么办

有三种方法来处理这一问题：

- 1) 不用它，避免初始化时的互相依赖。这是最好的解决方法。
- 2) 如果实在要用，就把那些关键的静态对象的定义放在一个文件中，这样只要让它们在文件中顺序正确就可以保证它们正确的初始化。
- 3) 如果确信把静态对象放在几个不同的翻译单元中是不可避免的——如在编写一个库时，这时无法控制那些使用该库的程序员——这可以通过两种程序设计技术加以解决。

##### 10.4.1.1 技术一

这是由Jerry Schwarz在创建*iostream*库（因为**cin**、**cout**和**cerr**是静态的且定义在不同的文件中）时首创的一种技术。它实际上没有第二种技术好，但是因为它的生存期比较长，这样可能会遇到很多代码使用了它。知道它的工作原理还是很重要的。

这一技术要求在库头文件中加上一个额外的类。这个类负责库中的静态对象的动态初始化。下面是一个简单的例子：

```
//: C10:Initializer.h
// Static initialization technique
#ifndef INITIALIZER_H
#define INITIALIZER_H
#include <iostream>
extern int x; // Declarations, not definitions
extern int y;

class Initializer {
    static int initCount;
public:
    Initializer() {
        std::cout << "Initializer()" << std::endl;
        // Initialize first time only
        if (initCount++ == 0) {
            std::cout << "performing initialization"
                      << std::endl;

            x = 100;
            y = 200;
        }
    }
    ~Initializer() {
```



```

        std::cout << "~Initializer()" << std::endl;
        // Clean up last time only
        if(--initCount == 0) {
            std::cout << "performing cleanup"
                      << std::endl;
            // Any necessary cleanup here
        }
    }
};

// The following creates one object in each
// file where Initializer.h is included, but that
// object is only visible within that file:
static Initializer init;
#endif // INITIALIZER_H ///:~

```

**x**、**y**的声明只是表明这些对象的存在，并没有为它们分配存储空间。然而**initializer init**的定义为每个包含此头文件的文件分配那些对象的存储空间，因为名字是**static**的（这里控制可见性而不是指定存储类型，因为默认时是在文件作用域内）它只在本翻译单元可见，所以连接器不会报告一个多重定义错误。

下面是一个包含**x**、**y**和**init\_Count**定义的文件：

```

//: C10:InitializerDefs.cpp {0}
// Definitions for Initializer.h
#include "Initializer.h"
// Static initialization will force
// all these values to zero:
int x;
int y;
int Initializer::initCount;
///:~

```

（当然，当一个文件包含头文件时，它的**init**静态实例也放在该文件中。）假设库的使用者产生了两个其他的文件：

```

//: C10:Initializer.cpp {0}
// Static initialization
#include "Initializer.h"
///:~

```

以及

```

//: C10:Initializer2.cpp
//{L} InitializerDefs Initializer
// Static initialization
#include "Initializer.h"
using namespace std;

int main() {
    cout << "inside main()" << endl;
    cout << "leaving main()" << endl;
} ///:~

```

现在哪个翻译单元先初始化都没有关系。当第一次包含**Initializer.h**的翻译单元被初始化时，**initCount**为零，这时初始化就已经完成了（这是由于任何动态初始化进行之前，静态存

储区已被设置为零)。对其余的翻译单元，`initCount`不会为零，并忽略初始化操作。清除将按相反的顺序发生，且`~Initializer()`可确保它只发生一次。

这个例子用内建类型作为全局静态对象，这种方法也可以用于类，但其对象必须用`initializer`类动态初始化。一种方法就是创建一个没有构造函数和析构函数的类，而是带有不同名字的用于初始化和清除的成员函数。当然更常用的做法是在`initializer()`函数中，设定有指向对象的指针，用`new`创建它们。

#### 10.4.1.2 技术二

在技术一使用很久之后才有人（我不知道是谁）提出了本小节将要说明的技术二。与技术一相比，这种技术更简单，也更清晰。之所以在技术一出现这么久之后才有技术二是因为C++太复杂。

这一技术基于这样的事实：函数内部的静态对象在函数第一次被调用时初始化，且只被初始化一次。需要记住的是，在这里真正想要解决的不是静态对象什么时候被初始化(这可以个别地加以控制)，而是确保正确的初始化顺序。

这种技术很灵巧。对于任何初始化依赖因素来说，可以把一个静态对象放在一个能返回对象引用的函数中。使用这种方法，访问静态对象的惟一途径就是调用这个函数。如果该静态对象需要访问其他依赖于它的静态对象时，就必须调用那些对象的函数。函数第一次被调用时，它强迫初始化发生。静态初始化的正确顺序是由设计的代码而不是由连接器任意指定顺序来保证的。

为了给出一个例子，这里有两个相互依赖的类。第一个类包含一个`bool`类型的成员，它只由构造函数初始化，所以能够知道该类的一个静态实例是否调用了构造函数（在程序开始时，静态存储区被初始化为零，如果没有调用构造函数的话，会对`bool`成员产生一个`false`值）。

```
//: C10:Dependency1.h
#ifndef DEPENDENCY1_H
#define DEPENDENCY1_H
#include <iostream>

class Dependency1 {
    bool init;
public:
    Dependency1() : init(true) {
        std::cout << "Dependency1 construction"
                  << std::endl;
    }
    void print() const {
        std::cout << "Dependency1 init: "
                  << init << std::endl;
    }
};
#endif // DEPENDENCY1_H ///:~
```

构造函数也显示它是什么时候被调用的，为了知道对象是否被初始化，可以通过`print()`函数打印出对象的状态。

第二个类初始化由第一个类的一个对象来完成，这将会导致初始化相互依赖。

```
//: C10:Dependency2.h
#ifndef DEPENDENCY2_H
```

```

#define DEPENDENCY2_H
#include "Dependency1.h"

class Dependency2 {
    Dependency1 d1;
public:
    Dependency2(const Dependency1& dep1): d1(dep1){
        std::cout << "Dependency2 construction ";
        print();
    }
    void print() const { d1.print(); }
};
#endif // DEPENDENCY2_H ///:~

```

构造函数显示它自己并打印出对象**d1**的状态，所以能够知道当构造函数被调用时，**d1**是否已经初始化了。

为了说明会出现什么错误，下面的文件首先以一种不正确的顺序定义静态对象，如果在对象**Dependency1**之前连接器碰巧初始化对象**Dependency2**，错误就会出现。如果定义的顺序恰好正确，那么就会以相反的顺序的显示说明它是如何正常工作的。这样，说明技术二是可靠的。

为了有更多的可读性的输出，增加**separator()**函数。诀窍就是不能全局地调用一个函数，除非该函数用来执行一个变量的初始化操作，所以**separator()**函数返回一个哑元值用来初始化两个全局变量。

```

//: C10:Technique2.cpp
#include "Dependency2.h"
using namespace std;

// Returns a value so it can be called as
// a global initializer:
int separator() {
    cout << "-----" << endl;
    return 1;
}

// Simulate the dependency problem:
extern Dependency1 dep1;
Dependency2 dep2(dep1);
Dependency1 dep1;
int x1 = separator();

// But if it happens in this order it works OK:
Dependency1 dep1b;
Dependency2 dep2b(dep1b);
int x2 = separator();

// Wrapping static objects in functions succeeds
Dependency1& d1() {
    static Dependency1 dep1;
    return dep1;
}

Dependency2& d2() {

```



```

    static Dependency2 dep2(d1());
    return dep2;
}
int main() {
    Dependency2& dep2 = d2();
} ///:~

```

函数d1()和d2()包含类Dependency1和Dependency2的静态对象。现在，访问这些静态对象的惟一方法就是调用这两个函数，并在第一次函数调用时强迫进行静态初始化，这可以保证初始化的正确性，通过这种方法，可以知道程序什么时候运行以及输出什么结果。

下面的代码使用了技术二。通常，静态对象在单独的文件中定义（由于某些原因，必须这样做；不过要记住在单独的文件中定义静态对象也会出现问题），而不是在单独的文件中定义一个包含静态对象的函数。但是需要在头文件中声明。

```

//: C10:Dependency1StatFun.h
#ifndef DEPENDENCY1STATFUN_H
#define DEPENDENCY1STATFUN_H
#include "Dependency1.h"
extern Dependency1& d1();
#endif // DEPENDENCY1STATFUN_H ///:~

```

实际上，关键字“extern”对于函数声明来说是多余的。下面是第二个头文件：

```

//: C10:Dependency2StatFun.h
#ifndef DEPENDENCY2STATFUN_H
#define DEPENDENCY2STATFUN_H
#include "Dependency2.h"
extern Dependency2& d2();
#endif // DEPENDENCY2STATFUN_H ///:~

```

在前面的实现文件中，有静态对象定义，现在，改为在包装的函数定义中定义静态对象：

```

//: C10:Dependency1StatFun.cpp {0}
#include "Dependency1StatFun.h"
Dependency1& d1() {
    static Dependency1 dep1;
    return dep1;
} ///:~

```

其他的代码也可以放在这些头文件中，下面是另外一个文件：

```

//: C10:Dependency2StatFun.cpp {0}
#include "Dependency1StatFun.h"
#include "Dependency2StatFun.h"
Dependency2& d2() {
    static Dependency2 dep2(d1());
    return dep2;
} ///:~

```

现在有两个文件，这两个文件可以以任意的顺序连接。如果它们只包含普通的静态对象，那么可以产生任意顺序的初始化。在这里因为它们包含定义静态对象的函数，所以不会出现不正确的初始化：

```

//: C10:Technique2b.cpp
//{L} Dependency1StatFun Dependency2StatFun

```

```
#include "Dependency2StatFun.h"
int main() { d2(); } ///:~
```

当运行这个程序时，将会发现**Dependency1**类的静态对象的初始化总是发生在类**Dependency2**的静态对象的初始化之前。所以，从中可以看出这种方法要比第一种技术简单得多。

我们也许想在函数**d1()**和**d2()**的头文件中把它们声明为内联函数，但是我们必须明确地知道这样做不行。内联函数在它出现的每一个文件中都会有一份副本——这种副本包括静态对象的定义。因为内联函数自动地默认为内部连接，所以这将导致多个重复的静态对象，且它们作用域为多个编译单元，这当然会出现问题。所以必须确保每一个定义了静态对象的函数只有一份定义，这就意味着不能把定义了静态对象的函数作为内联函数。

## 10.5 替代连接说明

如果在C++中编写一个程序需要用到C的库，那该怎么办呢？如果这样声明一个C函数：

```
float f(int a, char b);
```

C++的编译器就会将这个名称变成像**\_f\_int\_char**之类的东西以支持函数重载（和类型安全连接）。然而，C编译器编译的库一般不做这样的转换，所以它的内部名为**\_f**。这样，连接器将无法解释C++对**f()**的调用。

C++中提供了一个替代连接说明（*alternate linkage specification*），它是通过重载**extern**关键字来实现的。**extern**后跟一个字符串来指定想声明的函数的连接类型，后面是函数声明。

```
extern "C" float f(int a, char b);
```

这就告诉编译器**f()**是C连接，这样就不会转换函数名。标准的连接类型指定符有“C”和“C++”两种，但编译器开发商可选择用同样的方法支持其他语言。

如果有一组替代连接的声明，可以把它们放在花括号内：

```
extern "C" {
    float f(int a, char b);
    double d(int a, char b);
}
```

或在头文件中：

```
extern "C" {
    #include "Myheader.h"
}
```

多数C++编译器开发商在他们的头文件中处理转换连接指定，包括C和C++，所以不用担心它们。

## 10.6 小结

**static**关键字很容易使人糊涂，因为有时它控制存储分配，而有时控制一个名字的可见性和连接。

随着C++名字空间的引入，我们有了更好的、更灵活的方法来控制一个大项目中名字的增长。

在类的内部使用**static**是在全程序中控制名字的另一种方法。这些名字不会与全局名冲突，并且可见性和访问也限制在程序内部，使得在维护代码时能有更多的控制。

## 10.7 练习

部分练习题的答案可以在本书的电子文档“*Annotated Solution Guide for Thinking in C++*”中找到，只需支付很少的费用就可以从<http://www.BruceEckel.com>得到这个电子文档。

- 10-1 创建一个函数（带一个默认值为零的参数），函数内有一个静态变量的，这个静态变量是一个指针。当调用者为这个参数提供值时，它就指向一个整形数组的起始地址。如果用默认的参数值调用该函数，那么这个函数就返回数组的下一个值，直到它访问到数组中的“-1”（在数组中，-1作为结束的标志），在函数**main()**中调用这个函数。
- 10-2 创建一个这样的函数：每调用一次，它就返回Fibonacci序列中的下一个值。增加一个**bool**类型的参数，其默认值为**false**，当传递给该参数的值为**true**时重置函数使它指向Fibonacci序列的开头。在函数**main()**中调用这个函数。
- 10-3 创建一个有一个整型数组的类。在类内部用静态整型常量设置数组的大小。增加一个**const int** 变量，并在构造函数初始化列表中初始化。构造函数是内联的。增加一个**static int** 成员变量并用特定值来初始化。增加一个内联的成员函数**print()**，它打印数组中所有数组元素值并调用静态成员函数。在**main()**中运用这样的类。
- 10-4 创建一个类**Monitor**，它能知道它的成员函数**incident()**被调用了多少次。增加一个成员函数**print()**显示**incident()**被调用的次数，再创建一个包含一个静态的**Monitor**类的对象的函数。每次调用该函数时，它都会调用**print()**成员函数显示**incident()**被调用的次数。在主函数**main()**中调用这个函数。
- 10-5 修改练习4中的**Monitor**类，使其成员函数**decrement()**被调用时会减少记数。另创建一个类**Monitor2**，它的构造函数有一个指向**Monitor1**的指针参数，该构造函数存储指针值，调用**incident()**以及**print()**。**Monitor2**的析构函数调用**decrement()**和**print()**。写一个函数，在该函数中创建一个**Monitor2**的静态对象。在**main()**中测试调用该函数和不调用该函数时，**Monitor2**的析构函数各会出现什么结果。
- 10-6 定义一个**Monitor2**类的全局对象，看看会得到什么结果。
- 10-7 创建一个类，它的析构函数打印信息并调用**exit()**，定义该类的一个全局对象，看看会得到什么结果。
- 10-8 在文件**StaticDestructors.cpp**中，在**main()**内用不同的顺序调用**f()**、**g()**来检验构造函数与析构函数的调用顺序，你的编译器能正确地编译它们吗？
- 10-9 在文件**StaticDestructors.cpp**中，把**out**的最初定义变为一个**extern**声明，并把实际定义放到**a**（它的**Obj**构造函数传送信息给**out**）的定义之后，看看默认的错误处理是怎样工作的。运行程序时应确保没有其他重要程序在运行，否则机器会出现错误。
- 10-10 验证当带有多个静态变量的头文件被多个**cpp**文件包含时，不会有名字冲突。
- 10-11 创建一个简单的类，它包含一个整型数据成员，一个用自身参数初始化该数据成员的构造函数，还有一个用自身参数设置该成员值的成员函数，以及打印该成员值的**print()**函数。把该类放到头文件中去，在两个**cpp**文件中包含该头文件，在一个头文件中创建类的一个实例，在另外一个类中用**extern**声明，并在**main()**中测试。记住必须连接两个



对象文件，否则连接器将找不到所要连接的目标。

- 10-12 创建练习11中的类的静态实例，并验证：由于不存在**this**指针，连接器找不到它。
- 10-13 在一个头文件中声明一个函数。在另一个**cpp**文件中定义它，在第二个**cpp**文件的**main()**中调用这个函数，编译并验证它能正常运行。然后改变函数的定义，使它变为静态，验证连接器将找不到这个函数。
- 10-14 修改第8章中的**Volatile.cpp**文件，使**comm::isr()**能够像中断服务例程一样运行。注意：中断服务例程不带任何参数。
- 10-15 写一个使用**auto**和**register**关键字的简单程序，然后编译它。
- 10-16 创建一个包含一个名字空间的头文件。在名字空间里声明几个函数。再创建另一个头文件，它包含第一个头文件，并在先前的名字空间的基础上再增加几个函数声明。写一个包含第二个头文件的**cpp**文件。把名字空间用一个短的别名代替。在函数定义里使用作用域运算符调用这些函数。在另外一个单独的函数里，通过**using**指令把名字空间引入到函数中。并证实：这时并不需要作用域运算符调用名字空间里的函数。
- 10-17 创建一个带无名的名字空间的头文件。在两个单独的**cpp**文件中包含这个头文件，验证这个无名的名字空间对于这两个翻译单元来说都是一致的。
- 10-18 使用练习17的头文件，验证无名名字空间中的名字在一个翻译单元里即使不加指定也是可见的。
- 10-19 修改**FriendInjection.cpp**文件，增加一个友元函数的定义，在主函数**main()**中调用它。
- 10-20 在文件**Arithmetic.cpp**中，说明在一个函数中使用的**using**指令并不能扩展到这个函数的范围之外。
- 10-21 修改文件**OverridingAmbiguity.cpp**，先使用作用域运算符，然后用**using**声明代替作用域运算符来强迫编译器选择其中某个同名的函数名。
- 10-22 在两个头文件中，创建两个名字空间，每一个名字空间都包含一个类，且类名相同。创建一个包含这两个头文件的**cpp**文件。定义一个函数，在该函数中用**using**指令引入两个名字空间，然后创建类的一个对象，看看会有什么发生。再改变**using**指令的使用，使它为全局使用（在函数之外），看看结果是否不同。另外再使用作用域运算符，并创建两个类的对象。
- 10-23 用**using**声明修改练习22的程序，强迫编译器选择其中某个同名的类名。
- 10-24 去掉文件**BobsSuperDuperLibrary.cpp**和**UnnamedNamespaces.cpp**中的名字空间声明，把这些声明放到一个单独的头文件中，在处理过程中给这个无名的名字空间一个名字。在第三个文件中创建一个新的名字空间，该名字空间使用**using**声明包含其他两个名字空间。在主函数**main()**中使用**using**指令引用这个新的名字空间并访问所有的名字空间。
- 10-25 创建一个包含**<string>**和**<iostream>**的头文件，但不使用任何**using**指令和**using**声明。就像本书中所看到的一样，这里使用“**include**”。创建一个带有内联函数的类，它含有一个**string**成员，一个用自身参数初始化该成员的构造函数，还含有一个**print()**函数，它显示**String**成员的值，写一个**cpp**文件并在**main()**中运用这个类。
- 10-26 创建一个带**static double**和**long**类型的成员的类，写一个静态的成员函数并打印出这些静态数据成员的值。

- 10-27 创建一个类，它包含一个整型数据成员，一个通过自身参数初始化该整型数据成员的构造函数，还有一个显示这个整型数据成员的`print()`函数。再创建一个类，它包含第一个类的静态对象，增加一个静态成员函数并调用这个静态对象的`print()`函数，在主函数`main()`中运用这个类。
- 10-28 创建一个类，包含常量的和非常量的静态整型数组。写静态的方法来打印这些数组的值。在`main()`函数中运用这些类。
- 10-29 创建一个类，它包含一个`string`类型的数据成员，一个通过自身参数初始化该数据成员的构造函数，还有一个显示这个数据成员的`print()`函数，再创建一个类，它包含第一个类的对象的`const`和非`const`的静态对象数组，还有打印这些数组的静态方法。在`main()`函数中运用第二个类。
- 10-30 创建一个带整型成员和一个默认构造函数的结构(`struct`)，默认的构造函数把整型成员初始化为零。让这个结构局部于一个函数。在该函数中，创建一个该结构的对象数组，并演示这个数组中的整型被自动初始化为零。
- 10-31 创建一个类，它体现指针连接，但只允许使用一个指针。
- 10-32 在一个头文件中，创建一个类**Mirror**，它包含两个数据成员：一个是指向**Mirror**对象的指针和一个`bool`类型的数据成员，写两个构造函数：一个是默认的构造函数，它把`bool`成员初始化为`true`，使**Mirror**指向零值。第二个构造函数带有一个指向**Mirror**对象的指针参数，用该参数给对象的指针赋值。并把`bool`类型的数据成员设置成`false`。再增加一个成员函数`test()`，如果对象的指针成员为非零，则通过指针调用`test()`并返回它的值。如果指针是零，就返回`bool`类型的数据成员的值。然后写5个`cpp`文件，每一个都包含**Mirror**头文件。第一个`cpp`文件通过使用默认构造函数定义一个全局的**Mirror**对象。第二个`cpp`文件把第一个文件中定义的对象声明为`extern`，并通过使用第二个构造函数定义一个全局的**Mirror**对象，用一个指针指向第一个对象，在第三、四、五个文件中也做同样的处理。在最后一个文件中当然也包括一个全局对象的定义，并且`main()`应该调用`test()`函数并报告结果。如果结果为`true`，找出该如何改变连接器的连接顺序来使返回的结果为`false`。
- 10-33 用本书中介绍的技术一修改练习32中的程序。
- 10-34 用本书中介绍的技术二修改练习32中的程序。
- 10-35 写一个不包含任何头文件的程序，用标准的C库函数声明`puts()`，在主函数`main()`中调用这个函数。



## 引用和拷贝构造函数

引用就像是能自动地被编译器间接引用的常量型指针。

虽然引用Pascal语言中也有，但C++中引用的思想来自于Algol语言。在C++中，引用是支持运算符重载语法的基础（见第12章），也为函数参数的传入和传出控制提供了便利。

本章首先简单地介绍一下C和C++的指针的差异，然后介绍引用。但本章的大部分内容将研究对于C++新手来说比较含混的问题：拷贝构造函数（*copy-constructor*）。它是一种特殊的构造函数，需要用引用来实现从现有的相同类型的对象中产生新的对象。编译器使用拷贝构造函数通过按值传递（*by value*）的方式在函数中传递和返回对象。

本章最后将阐述有点难以理解的C++的成员指针（*pointer-to-member*）这个概念。

### 11.1 C++中的指针

C和C++指针的最重要的区别在于C++是一种类型要求更强的语言。就`void*`而言，这一点表现得更加突出。C不允许随便地把一个类型的指针赋值给另一个类型，但允许通过`void*`来实现。例如：

```
bird* b;  
rock* r;  
void* v;  
v = r;  
b = v;
```

由于C的这种功能允许把任何一种类型看做别的类型处理，这就在类型系统中留下了一个大的漏洞。C++不允许这样做，其编译器将会给出一个出错信息。如果真想把某种类型当做别的类型处理，则必须显式地使用类型转换，通知编译器和读者（第3章已经介绍了C++的经过改进“显式”类型转换语法）。

### 11.2 C++中的引用

引用（*reference*）（&）就像能自动地被编译器间接引用的常量型指针。它通常用于函数的参数表中和函数的返回值，但也可以独立使用。例如：

```
//: C11:FreeStandingReferences.cpp  
#include <iostream>  
using namespace std;  
  
// Ordinary free-standing reference:  
int y;  
int& r = y;  
// When a reference is created, it must  
// be initialized to a live object.  
// However, you can also say:  
const int& q = 12; // (1)
```

```

// References are tied to someone else's storage:
int x = 0;           // (2)
int& a = x;          // (3)
int main() {
    cout << "x = " << x << ", a = " << a << endl;
    a++;
    cout << "x = " << x << ", a = " << a << endl;
} ///:~

```

在行 (1) 中，编译器分配了一个存储单元，它的值被初始化为12，于是这个引用就和这个存储单元联系上了。应用要点是任何引用必须和存储单元联系。访问引用时，就是在访问那个存储单元。因而，如果写行 (2) 和 (3)，那么增加a事实上就是增加x，这个可在main()函数中显示出来。思考一个引用的最简单的方法是把它当做一个奇特的指针。这个指针的一个优点是不必怀疑它是否被初始化了（编译器强迫它初始化），也不必知道怎样对它间接引用（这由编译器做）。

使用引用时有一定的规则：

- 1) 当引用被创建时，它必须被初始化（指针则可以在任何时候被初始化）。
- 2) 一旦一个引用被初始化为指向一个对象，它就不能改变为另一个对象的引用（指针则可以在任何时候指向另一个对象）。
- 3) 不可能有NULL引用。必须确保引用是和一块合法的存储单元关联。

### 11.2.1 函数中的引用

最经常看见引用的地方是在函数参数和返回值中。当引用被用做函数参数时，在函数内任何对引用的更改将对函数外的参数产生改变。当然，可以通过传递一个指针来做相同的事情，但引用具有更清晰的语法。（如果愿意的话，可以把引用看做一个使语法更加便利的工具。）

如果从函数中返回一个引用，必须像从函数中返回一个指针一样对待。当函数返回时，无论引用关联的是什么都应该存在，否则，将不知道指向哪一个内存。

下面有一个例子：

```

//: C11:Reference.cpp
// Simple C++ references

int* f(int* x) {
    (*x)++;
    return x; // Safe, x is outside this scope
}

int& g(int& x) {
    x++; // Same effect as in f()
    return x; // Safe, outside this scope
}

int& h() {
    int q;
    //! return q; // Error
    static int x;
    return x; // Safe, x lives outside this scope
}

```



```
int main() {
    int a = 0;
    f(&a); // Ugly (but explicit)
    g(a);  // Clean (but hidden)
} ///:~
```

对函数f()的调用缺乏使用引用的方便性和清晰性，但很清楚这是传递一个地址。在函数g()的调用中，地址通过引用被传递，但表面上看不出来。

#### 11.2.1.1 常量引用

仅当在**Reference.cpp**中的参数是非常量对象时，这个引用参数才能工作。如果是常量对象，函数g()将不接受这个参数，这样做是一件好事，因为这个函数将改变外部参数。如果知道这函数不妨碍对象的不变性的话，让这个参数是一个常量引用将允许这个函数在任何情况下使用。这意味着，对于内建类型，这个函数不会改变参数，而对于用户定义的类型，该函数只能调用常量成员函数，而且不应当改变任何公共的数据成员。

在函数参数中使用常量引用特别重要。这是因为我们的函数也许会接受临时对象，这个临时对象是由另一个函数的返回值创立或由函数使用者显式地创立的。临时对象总是不变的，因此如果不使用常量引用，参数将不会被编译器接受。看下面一个非常简单的例子：

```
//: C11:ConstReferenceArguments.cpp
// Passing references as const

void f(int&) {}
void g(const int&) {}

int main() {
    //! f(1); // Error
    g(1);
} ///:~
```

调用f(1)会产生编译期间错误，这是因为编译器必须首先建立一个引用，即编译器为一个int类型分派存储单元，同时将其初始化为1并为其产生一个地址和引用捆绑在一起。存储的内容必须是常量，因为改变它没有任何意义——我们再不能对它进行操作。对于所有的临时对象，必须同样假设它们是不可存取的。当改变这种数据的时候，编译器会指出错误，这是非常有用的提示，因为这个改变会导致信息丢失。

#### 11.2.1.2 指针引用

在C语言中，如果想改变指针本身而不是它所指向的内容，函数声明可能像这样：

```
void f(int**);
```

当传递它时，必须取得指针的地址：

```
int i = 47;
int* ip = &i;
f(&ip);
```

对于C++中的引用，语法清晰多了。函数参数变成指针的引用，用不着取得指针的地址。因此，

```
//: C11:ReferenceToPointer.cpp
#include <iostream>
```

```
using namespace std;

void increment(int*& i) { i++; }

int main() {
    int* i = 0;
    cout << "i = " << i << endl;
    increment(i);
    cout << "i = " << i << endl;
} ///:~
```

通过运行这个程序，将会看到指针本身增加了，而不是它指向的内容增加了。

### 11.2.2 参数传递准则

当给函数传递参数时，人们习惯上是通过常量引用来传递。虽然最初看起来似乎仅是出于效率考虑（通常在设计和装配程序时并不考虑效率），但像本章以后部分介绍的，这里将会带来很多的危险。拷贝构造函数需要通过传值方式来传递对象，但这并不总是可行的。

这种简单习惯可以大大提高效率：传值方式需要调用构造函数和析构函数，然而如果不想改变参数，则可通过常量引用传递，它仅需要将地址压栈。

事实上，只有一种情况不适合用传递地址方式，这就是当传值是惟一安全的途径，否则将会破坏对象时（不想修改外部对象，这不是调用者通常期望的）。这是下一节的主题。

## 11.3 拷贝构造函数

介绍了C++中引用的基本概念后，我们将讲述一个更令人混淆的概念：拷贝构造函数，它常被称为**X(X&)**（“X引用的X”）。在函数调用时，这个构造函数是控制通过传值方式传递和返回用户定义类型的根本所在。事实上，我们将会看到，这是很重要的，以至于编译器在没有提供拷贝构造函数时将会自动地创建。

### 11.3.1 按值传递和返回

为了理解拷贝构造函数的需要，看一下C语言在调用函数时处理通过按值传递和返回变量的方法。如果声明了一个函数并调用它：

```
int f(int x, char c);
int g = f(a, b);
```

编译器如何知道怎样传递和返回这些变量？其实它天生就知道！因为它必须处理的类型的范围是如此之小（**char**、**int**、**float**、**double**和它们的变量），这些信息都被内置在编译器中。

如果能了解编译器怎样产生汇编代码和确定调用函数**f()**而产生的语句，以上语句就相当于：

```
push b
push a
call f()
add sp,4
mov g, register a
```

这个代码已被认真整理过，使之具有普遍意义；**b**和**a**的表达式根据变量是全局变量（在

这种情况下它们是**\_b**和**\_a**)或局部变量(编译器将在堆栈上对其索引)将有差异。**g**表达式也是这样。对**f()**调用的形式取决于名字修饰表,“寄存器**a**”取决于CPU寄存器在汇编程序中是如何命名的。但不管代码如何,逻辑是相同的。

在C和C++中,参数是从右向左进栈的,然后调用函数,调用代码负责清理栈中的参数(这一点说明了**add sp,4**的作用)。但是要注意,通过按值传递方式传递参数时,编译器简单地将参数拷贝压栈——编译器知道拷贝有多大,并知道如何对参数压栈,对它们正确地拷贝。

**f()**的返回值放在寄存器中。编译器同样知道返回值的类型,因为这个类型是内置于语言中的,于是编译器可以通过把返回值放在寄存器中返回它。在C的基本数据类型中,拷贝这个值的位的行为就等同于拷贝这个对象。

#### 11.3.1.1 传递和返回大对象

现在来考虑用户定义的类型。如果创建了一个类,希望通过传值方式传递该类的一个对象,编译器怎样知道做什么?这是编译器所不知的非内建数据类型,是别人创建的类型。

为了研究这个问题,首先从一个简单的结构开始,这个结构太大以至于不能在寄存器中返回:

```
//: C11:PassingBigStructures.cpp
struct Big {
    char buf[100];
    int i;
    long d;
} B, B2;

Big bigfun(Big b) {
    b.i = 100; // Do something to the argument
    return b;
}

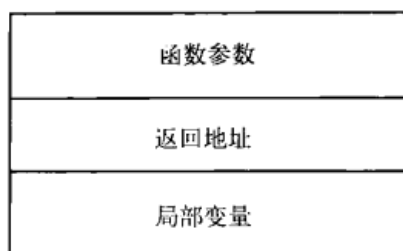
int main() {
    B2 = bigfun(B);
} ///:~
```

在这里列出汇编代码有点复杂,因为大多数编译器使用辅助(helper)函数而不是简单插入功能性的语句。在**main()**函数中,正如我们猜测的,首先调用函数**bigfun()**,整个**B**的内容被压栈(我们可能发现有些编译器把**B**的地址和大小装入寄存器,然后调用辅助函数把它压栈)。

在先前的例子中,调用函数之前要把参数压栈。然而,在**PassingBigStructures.cpp**中,将看到附加的操作:在函数调用之前,**B2**的地址压栈,虽然它明显不是一个参数。为了理解这里发生的事,必须了解当编译器调用函数时对编译器的约束。

#### 11.3.1.2 函数调用栈框架

当编译器为函数调用产生代码时,它首先把所有的参数压栈,然后调用函数。在函数内部,产生代码,向下移动栈指针为函数局部变量提供存储单元。(在这里“下”是相对的,在压栈时,机器的栈指针可能增加也可能减小。)但是在汇编语言CALL中,CPU把程序代码中的函数调用指令的地址压栈,所以汇编语言RETURN可以使用这个地址返回到调用点。当然,这个地址是非常重要的,因为没有它程序将迷失方向。这里提供一个在CALL后栈框架的样子,此时在函数中已为局部变量分配了存储单元。



函数的其他部分产生的代码希望能完全按照这个方法安排内存，因此它可以谨慎地从函数参数和局部变量中存取而不触及返回地址。称在函数调用过程中被函数使用的这块内存为函数框架(*function frame*)。

另外，试图从栈中得到返回值是合理的。因为编译器简单地把返回值压栈，函数可以返回一个偏移值，它告诉返回值的开始在栈中所处的位置。

#### 11.3.1.3 重入

因为在C和C++中的函数支持中断，所以这将出现语言重入的难题。同时，它们也支持函数递归调用。这就意味着在程序执行的任何时候，中断都可以发生而不打乱程序。当然，编写中断服务程序(ISR)的作者负责存储和还原所使用的所有的寄存器(可以把ISR看成没有参数和返回值是**void**的普通函数，它存储和还原CPU的状态。有些硬件事件触发一个ISR函数的调用，而不是在程序中显式地调用)。

现在来想象一下，如果普通函数试着在堆栈中返回值，将会发生什么。因为不能触及堆栈返回地址以上任何部分，所以函数必须在返回地址以下将值压栈。但当汇编语言RETURN执行时，堆栈指针必须指向返回地址(或正好位于它下面，这取决于机器)，所以恰好在RETURN语句之前，函数必须将堆栈指针向上移动，这便清除了所有局部变量。但如果试图从堆栈中的返回地址下返回数值，因为中断可能此时发生，此时是最易被攻击的时候。这个时候ISR将向下移动堆栈指针，保存返回地址和局部变量，这样就会覆盖掉返回值。

为了解决这个问题，在调用函数之前，调用者应负责在堆栈中为返回值分配额外的存储单元。然而，C不是按照这种方法设计的，C++也一样。正如不久将看到的，C++编译器使用更有效的方案。

下一个想法可能是在全局数据区域返回数值，但这不可行。重入意味着任何函数可以中断任何其他函数，包括当前所处的相同函数。因此，如果把返回值放在全局区域，可能又返回到相同的函数中，这将重写返回值。对于递归也是同样的道理。

惟一安全的返回场所是寄存器，问题是当寄存器没有用于存放返回值的足够大小时该怎么做。答案是把返回值的地址像一个函数参数一样压栈，让函数直接把返回值信息拷贝到目的地。这也是在**PassingBigStructures.cpp**的**main()**中**bigfun()**调用之前将B2的地址压栈的原因。如果看了**bigfun()**的汇编输出，可以看到它接收这个隐藏的参数并在函数内完成向目的地的拷贝。

#### 11.3.1.4 位拷贝与初始化

迄今为止，一切都很顺利。对于传递和返回大的简单结构有了可使用的方法。但注意所用的方法是从一个地方向另一个地方拷贝位，这对于C考虑的变量的原始方法当然进行得很好。但在C++中，对象比一组比特位要复杂得多，因为对象具有含义。这个含义也许不能由它具有的位拷贝来很好地反映。



下面来考虑一个简单的例子：一个类在任何时候都知道它存在多少个对象。从第10章了解到可以通过包含一个静态数据成员的方法来做到这点。

```

//: C11:HowMany.cpp
// A class that counts its objects
#include <fstream>
#include <string>
using namespace std;
ofstream out("HowMany.out");

class HowMany {
    static int objectCount;
public:
    HowMany() { objectCount++; }
    static void print(const string& msg = "") {
        if(msg.size() != 0) out << msg << ": ";
        out << "objectCount = "
            << objectCount << endl;
    }
    ~HowMany() {
        objectCount--;
        print("~HowMany()");
    }
};

int HowMany::objectCount = 0;

// Pass and return BY VALUE:
HowMany f(HowMany x) {
    x.print("x argument inside f()");
    return x;
}

int main() {
    HowMany h;
    HowMany::print("after construction of h");
    HowMany h2 = f(h);
    HowMany::print("after call to f()");
} ///:~

```

**HowMany**类包括一个静态变量`int objectCount`和一个用于报告这个变量的静态成员函数`print()`，这个函数有一个可选择的消息参数。每当一个对象产生时，构造函数增加记数，而对象销毁时，析构函数减小记数。

然而，输出并不是所期望的那样：

```

after construction of h: objectCount = 1
x argument inside f(): objectCount = 1
~HowMany(): objectCount = 0
after call to f(): objectCount = 0
~HowMany(): objectCount = -1
~HowMany(): objectCount = -2

```

在`h`生成以后，对象数是1，这是对的。我们希望在`f()`调用后对象数是2，因为`h2`也在范围内。然而，对象数是0，这意味着发生了严重的错误。这从结尾两个析构函数执行后使得对象数变为负数的事实得到确认，有些事根本就不应该发生。

让我们来看一下函数 `f()` 通过按值传递方式传入参数那一处。原来的对象 `h` 存在于函数框架之外，同时在函数体内又增加了一个对象，这个对象是通过传值方式传入的对象的拷贝。然而，参数的传递是使用 C 的原始的位拷贝的概念，但 C++ `HowMany` 类需要真正的初始化来维护它的完整性。所以，默认的位拷贝不能达到预期的效果。

在对 `f()` 的调用的最后，当局部对象出了其范围时，析构函数就被调用，析构函数使 `objectCount` 减小。所以，在函数外面，`objectCount` 等于 0。`h2` 对象的创建也是用位拷贝产生的，所以，构造函数在这里也没有调用。当对象 `h` 和 `h2` 出了它们的作用范围时，它们的析构函数就使 `objectCount` 值变为负值。

### 11.3.2 拷贝构造函数

出现上述问题是因为编译器对如何从现有的对象产生新的对象进行了假定。当通过按值传递的方式传递一个对象时，就创立了一个新对象，函数体内的对象是由函数体外的原来存在的对象传递的。从函数返回对象也是同样的道理。在表达式中：

```
HowMany h2 = f(h);
```

先前未创立的对象 `h2` 是由函数 `f()` 的返回值创建的，所以又从一个现有的对象中创建了一个新对象。

编译器假定我们想使用位拷贝来创建对象。在许多情况下，这是可行的。但在 `HowMany` 类中就行不通，因为初始化不是简单的拷贝。如果类中含有指针又将出现另一个问题：它们指向什么内容，是否拷贝它们或它们是否与一些新的内存块相连？

幸运的是，可以介入这个过程，并可以防止编译器进行位拷贝。每当编译器需要从现有的对象创建新对象时，可以通过定义自己的函数做这些事。因为是在创建新对象，所以，这个函数应该是构造函数，并且传递给这个函数的单一参数必须是创立的对象的源对象。但是这个对象不能通过按值传递方式传入构造函数，因为正在试图定义的函数就是为了处理按值传递方式的，而且按句法传递一个指针是没有意义的，毕竟我们正在从现有的对象创建新对象。这里，引用就起作用了，可以使用源对象的引用。这个函数被称为拷贝构造函数，它经常被称为 `X(X&)`（它叫做类 `X` 的外在表现）。

如果设计了拷贝构造函数，当从现有的对象创建新对象时，编译器将不使用位拷贝。编译器总是调用我们的拷贝构造函数。所以，如果没有设计拷贝构造函数，编译器将做一些判断，但可以选择完全接管这个过程的控制。

现在可以解决 `HowMany.cpp` 中的问题。

```
//: C11:HowMany2.cpp
// The copy-constructor
#include <fstream>
#include <string>
using namespace std;
ofstream out("HowMany2.out");

class HowMany2 {
    string name; // Object identifier
    static int objectCount;
public:
    HowMany2(const string& id = "") : name(id) {
```



```

        ++objectCount;
        print("HowMany2()");
    }
    ~HowMany2() {
        --objectCount;
        print("~HowMany2()");
    }
    // The copy-constructor:
    HowMany2(const HowMany2& h) : name(h.name) {
        name += " copy";
        ++objectCount;
        print("HowMany2(const HowMany2&)");
    }
    void print(const string& msg = "") const {
        if(msg.size() != 0)
            out << msg << endl;
        out << '\t' << name << ": "
            << "objectCount = "
            << objectCount << endl;
    }
};

int HowMany2::objectCount = 0;

// Pass and return BY VALUE:
HowMany2 f(HowMany2 x) {
    x.print("x argument inside f()");
    out << "Returning from f()" << endl;
    return x;
}

int main() {
    HowMany2 h("h");
    out << "Entering f()" << endl;
    HowMany2 h2 = f(h);
    h2.print("h2 after call to f()");
    out << "Call f(), no return value" << endl;
    f(h);
    out << "After call to f()" << endl;
} ///:~

```

这儿加入一些新的方法，使我们能很好地理解发生过程。首先，当对象的信息被打印出来时，**string name**起着对象识别作用。在构造函数内，可以设置一个标识符字符串（通常是对象的名字），它通过**string**构造函数拷贝至**name**中。默认值""构造了一个空字符串。同样，构造函数将增加而析构函数减少**objectCount**的值。

其次是拷贝构造函数**HowMany2(const HowMany2&)**。拷贝构造函数可以仅从现有的对象创立新对象，所以，现有的对象的名字被拷贝给**name**，这样就能了解它是从哪里拷贝来的。如果深入了解，将会看到在构造函数的初始化表上对**name(h.name)**的调用事实上就是调用了**string**拷贝构造函数。

在拷贝构造函数内部，对象数目会像普通构造函数一样的增加。这意味着当参数通过按值传递方式传递和返回时，我们能得到准确的对象数目。

**print()**函数已经被修改，用于打印消息、对象标识符和对象数目。现在**print()**函数必须

访问具体对象的**name**数据，所以不再是静态成员函数。

在**main()**函数内部，可以看到又增加了一次函数**f()**的调用。但这次使用了普通的C语言调用方式，且忽略了函数的返回值。既然现在知道了值是如何返回的（即在函数体内，代码处理返回过程并把结果放在目的地，目的地的地址作为一个隐藏的参数传递）。返回值被忽略将会发生什么，程序的输出将对此作出解释。

在显示输出之前，这里有一个小程序，它使用了**iostreams**可为任何文件加入行号。

```
//: C11:Linenum.cpp
//{T} Linenum.cpp
// Add line numbers
#include "../require.h"
#include <vector>
#include <string>
#include <fstream>
#include <iostream>
#include <cmath>
using namespace std;

int main(int argc, char* argv[]) {
    requireArgs(argc, 1, "Usage: linenum file\n"
        "Adds line numbers to file");
    ifstream in(argv[1]);
    assure(in, argv[1]);
    string line;
    vector<string> lines;
    while(getline(in, line)) // Read in entire file
        lines.push_back(line);
    if(lines.size() == 0) return 0;
    int num = 0;
    // Number of lines in file determines width:
    const int width = int(log10(lines.size())) + 1;
    for(int i = 0; i < lines.size(); i++) {
        cout.setf(ios::right, ios::adjustfield);
        cout.width(width);
        cout << ++num << " " << lines[i] << endl;
    }
} ///:~
```

整个文件被读入**vector<string>**，这使用了本书前面同样的代码。当打印行号时，我们希望所有的行都能彼此对齐，这就要求在文件中调整行的数目，以使得各行号所允许的宽度是一致的。我们可以轻松地通用**vector::size()**决定行的数目，但我们真正所需要知道的是它们是否超过了10行、100行、1000行等。如果对文件的行数取以10为底的对数，把它转为整型并再加1，这样就可得到行的最大宽度。

我们将会注意到，在**for**循环的内部有两个特殊的调用：**setf()**和**width()**。在这方面，**ostream**调用允许控制对齐方式和输出的宽度。但是它们必须在每一行被输出时都要调用，这也就是为什么它们被置于**for**循环内的原因。在本书的第2卷有一章是说明输出流的，它将介绍更多有关控制输出流的调用和其他的一些方法。

当**Linenum.cpp**被应用于**HowMany2.out**时，结果如下：

```
1) HowMany2()
2) h: objectCount = 1
```

```

3) Entering f()
4) HowMany2(const HowMany2&)
5)   h copy: objectCount = 2
6) x argument inside f()
7)   h copy: objectCount = 2
8) Returning from f()
9) HowMany2(const HowMany2&)
10)  h copy copy: objectCount = 3
11) ~HowMany2()
12)  h copy: objectCount = 2
13) h2 after call to f()
14)  h copy copy: objectCount = 2
15) Call f(), no return value
16) HowMany2(const HowMany2&)
17)  h copy: objectCount = 3
18) x argument inside f()
19)  h copy: objectCount = 3
20) Returning from f()
21) HowMany2(const HowMany2&)
22)  h copy copy: objectCount = 4
23) ~HowMany2()
24)  h copy: objectCount = 3
25) ~HowMany2()
26)  h copy copy: objectCount = 2
27) After call to f()
28) ~HowMany2()
29)  h copy copy: objectCount = 1
30) ~HowMany2()
31)  h: objectCount = 0

```

正如所希望的，第一件发生的事是为**h**调用普通的构造函数，对象数增加为1。但在进入函数**f()**时，拷贝构造函数被编译器调用，完成传值过程。在**f()**内创建了一个新对象，它是**h**的拷贝（因此被称为“**h**拷贝”），所以对象数变成2，这是拷贝构造函数的作用结果。

第8行显示了从**f()**返回的开始情况。但在局部变量“**h**拷贝”销毁以前（在函数结尾这个局部变量便出了范围），它必须被拷入返回值，也就是**h2**。先前未创建的对象（**h2**）是从现有的对象（在函数**f()**内的局部变量）创建的，所以在第9行拷贝构造函数当然又被使用。现在，对于**h2**的标识符，名字变成了“**h**拷贝的拷贝”。因为它是从拷贝拷过来的，这个拷贝是函数**f()**内部对象。在对象返回之后，函数结束之前，对象数暂时变为3，但此后内部对象“**h**拷贝”被销毁。在13行完成对**f()**的调用后，仅有2个对象**h**和**h2**。这时可以看到**h2**最终是“**h**拷贝的拷贝”。

#### 11.3.2.1 临时对象

第15行开始调用**f(h)**，这次调用忽略了返回值。在16行可以看到恰好在参数传入之前，拷贝构造函数被调用。和前面一样，21行显示了为了返回值而调用拷贝构造函数。但是，拷贝构造函数必须有一个作为它的目的地（**this**指针）的工作地址。但这个地址从哪里获得呢？

每当编译器为了正确地计算一个表达式而需要一个临时对象时，编译器可以创建一个。在这种情况下，编译器创建一个看不见的对象作为函数**f()**忽略了的返回值的目标地址。这个临时对象的生存期应尽可能的短，这样，空间就不会被这些等待被销毁且占用珍贵资源的临时对象搞乱。在一些情况下，临时对象可能立即传递给另外的函数。但在现在这种情况下，临时对象在函数调用之后不再需要，所以一旦函数调用完结就对内部对象调用析构函数（23

和24行), 这个临时对象就被销毁 (25和26行)。

在28-31行, 对象**h2**被销毁了, 接着对象**h**被销毁。对象记数非常正确地回到了0。

### 11.3.3 默认拷贝构造函数

因为拷贝构造函数实现按值传递方式的参数传递和返回, 所以在这种简单结构情况下, 编译器将有效地创建一个默认拷贝构造函数, 这非常重要。在C中也是这样。然而, 直到目前所看到的一切默认的都是原始行为: 位拷贝。

当包括更复杂的类型时, 如果没有创建拷贝构造函数, C++编译器也将自动地创建拷贝构造函数。然而, 又一次的, 位拷贝没有意义, 它并不能达到我们的意图。

这儿有一个例子显示编译器采取的更聪明的方法。设想创建了一个新类, 它是由某些现有类的对象组成的。这个创建类的方法被称为组合 (*composition*), 它是从现有类创建新类的方法之一。现在, 假设用这个方法快速创建一个新类来解决某个问题。因为还不知道拷贝构造函数, 所以没有创建它。下面的例子演示了当编译器为新类创建默认拷贝构造函数时, 编译器做了哪些事。

```

//: C11:DefaultCopyConstructor.cpp
// Automatic creation of the copy-constructor
#include <iostream>
#include <string>
using namespace std;

class WithCC { // With copy-constructor
public:
    // Explicit default constructor required:
    WithCC() {}
    WithCC(const WithCC&) {
        cout << "WithCC(WithCC&)" << endl;
    }
};

class WoCC { // Without copy-constructor
    string id;
public:
    WoCC(const string& ident = "") : id(ident) {}
    void print(const string& msg = "") const {
        if(msg.size() != 0) cout << msg << ": ";
        cout << id << endl;
    }
};

class Composite {
    WithCC withcc; // Embedded objects
    WoCC wocc;
public:
    Composite() : wocc("Composite()") {}
    void print(const string& msg = "") const {
        wocc.print(msg);
    }
};

int main() {

```



```

Composite c;
c.print("Contents of c");
cout << "Calling Composite copy-constructor"
    << endl;
Composite c2 = c; // Calls copy-constructor
c2.print("Contents of c2");
} ///:~

```

类**WithCC**有一个拷贝构造函数，这个函数只是简单地宣布它被调用，这引出了一个有趣的问题。在类**Composite**中，使用默认的构造函数创建一个**WithCC**类的对象。如果在类**WithCC**中根本没有构造函数，编译器将自动地创建一个默认的构造函数。不过在这种情况下，这个构造函数什么也不做。然而，如果加了一个拷贝构造函数，我们就告诉了编译器我们将自己处理构造函数的创建，编译器将不再创建默认的构造函数，并且，除非我们显式地创建一个默认的构造函数，就如同为类**WithCC**所做的那样，否则将指示出错。

类**WoCC**没有拷贝构造函数，但它的构造函数将在内部**string**中存储一个信息，这个信息可以使用**print()**函数打印出来。这个构造函数在类**Composite**构造函数的初始化表达式表（初始化表达式表已在第8章简单地介绍过了，并将在第14章中全面介绍）中被显式地调用。这样做的原因在稍后将会明白。

类**Composite**既含有**WithCC**类的成员对象又含有**WoCC**类的成员对象（注意因为必须如此做，内嵌的对象**WoCC**在构造函数初始化表中被初始化了）。类**Composite**没有显式定义的拷贝构造函数。然而，在**main()**函数中，按下面的定义使用拷贝构造函数创建了一个对象。

```
Composite c2 = c;
```

类**Composite**的拷贝构造函数由编译器自动创建，程序的输出显示了它是如何被创建的。

```

Contents of c: Composite()
Calling Composite copy-constructor
WithCC(WithCC&)
Contents of c2: Composite()

```

为了对使用组合（和继承的方法，将在第14章介绍）的类创建拷贝构造函数，编译器递归地为所有的成员对象和基类调用拷贝构造函数。如果成员对象还含有别的对象，那么后者的拷贝构造函数也将被调用。所以，在这里，编译器也为类**WithCC**调用拷贝构造函数。程序的输出显示了这个构造函数被调用。因为**WoCC**没有拷贝构造函数，编译器为它创建一个，该拷贝构造函数仅执行了位拷贝。编译器在类**Composite**的拷贝构造函数内部调用了这个刚创建的拷贝构造函数，这可由在**main**中调用**Composite::print()**显示出来，因为**c2.wocc**的内容与**c.wocc**内容是相同的。编译器获得一个拷贝构造函数的过程被称为成员方法初始化（*memberwise initialization*）。

最好的方法是创建自己的拷贝构造函数而不让编译器创建。这样就能保证程序在我们的控制之下。

#### 11.3.4 替代拷贝构造函数的方法

现在，我们可能头已发晕了。我们可能想，怎样才能不必了解拷贝构造函数就能写一个具有一定功能的类。但是别忘了：仅当准备用按值传递的方式传递类对象时，才需要拷贝构造函数。如果不那么做时，就不需要拷贝构造函数。

#### 11.3.4.1 防止按值传递

我们也许会说：“如果我自己不写拷贝构造函数，编译器将为我创建。所以，我怎么能保证一个对象将永远不会被通过按值传递方式传递呢？”

有一个简单的技术防止通过按值传递方式传递：声明一个私有拷贝构造函数。甚至不必去定义它，除非成员函数或友元函数需要执行按值传递方式的传递。如果用户试图用按值传递方式传递或返回对象，编译器将会发出一个出错信息。这是因为拷贝构造函数是私有的。因为已显式地声明我们接管了这项工作，所以编译器不再创建默认的拷贝构造函数。例如：

```
//: C11:NoCopyConstruction.cpp
// Preventing copy-construction

class NoCC {
    int i;
    NoCC(const NoCC&); // No definition
public:
    NoCC(int ii = 0) : i(ii) {}
};

void f(NoCC);

int main() {
    NoCC n;
    //! f(n); // Error: copy-constructor called
    //! NoCC n2 = n; // Error: c-c called
    //! NoCC n3(n); // Error: c-c called
} ///:~
```

注意使用的很普遍的形式

```
NoCC(const NoCC&);
```

这里使用了**const**。

#### 11.3.4.2 改变外部对象的函数

引用语法比指针语法好用，但对于读者来说，它却使意思变得模糊。例如，在*iostreams*库函数中，一个重载版函数**get()**是用一个**char&**作为参数，这个函数的作用是通过插入**get()**的结果而改变它的参数。然而，当阅读使用这个函数的代码时，我们不会立即明白外面的对象正被改变：

```
char c;
cin.get(c);
```

相反，此函数调用看起来更像是按值传递方式传递，暗示着外部对象没有被改变。

正因为如此，当传递一个可被修改的参数地址时，从代码维护的观点看，使用指针可能更安全些。如果总是应用**const**引用传递地址，除非打算通过地址修改外部对象（这个地址通过非**const**指针传递），这样读者更容易读懂我们的代码。

## 11.4 指向成员的指针

指针是指向一些内存地址的变量，既可以是数据的地址也可以是函数的地址。所以，可以在运行时改变指针指向的内容。C++的成员指针（*pointer-to-member*）遵从同样的概念，除



了所选择的内容是在类中之内的成员指针。这里麻烦的是所有的指针需要地址，但在类内部是没有地址的；选择一个类的成员意味着在类中偏移。只有把这个偏移和具体对象的开始地址结合，才能得到实际地址。成员指针的语法要求选择一个对象的同时间接引用成员指针。

为了理解这个语法，先来考虑一个简单的结构：如果有一个这样结构的指针`sp`和对象`so`，可以通过下面方法选择成员：

```
//: C11:SimpleStructure.cpp
struct Simple { int a; };
int main() {
    Simple so, *sp = &so;
    sp->a;
    so.a;
} ///:~
```

现在，假设有一个普通的指向`integer`的指针`ip`。为了取得`ip`指向的内容，用一个`*`号间接引用这个指针。

```
*ip = 4;
```

最后，考虑如果有一个指向一个类对象成员的指针，如果假设它代表对象内一定的偏移，将会发生什么？为了取得指针指向的内容，必须用`*`号间接引用。但是，它只是一个对象内的偏移，所以必须也要指定那个对象。因此，`*`号要和间接引用的对象结合。所以，对于指向一个对象的指针，新的语法变为`->*`，对于一个对象或引用，则为`.*`，如下所示。

```
objectPointer->*pointerToMember = 47;
object.*pointerToMember = 47;
```

现在，让我们看看定义`pointerToMember`的语法是什么？其实它像任何一个指针，必须说出它指向什么类型。并且，在定义中也要使用一个`*`号。惟一的区别只是它必须说出这个成员指针使用什么类的对象。当然，这是用类名和作用域运算符实现的。因此，可表示如下：

```
int ObjectClass::*pointerToMember;
```

定义一个名字为`pointerToMember`的成员指针，该指针可以指向在`ObjectClass`类中的任一`int`类型的成员。还可以在定义的时候初始化这个成员指针。

```
int ObjectClass::*pointerToMember = &ObjectClass::a;
```

因为仅仅提到了一个类而非那个类的对象，所以没有`ObjectClass::a`的确切“地址”。因而，`&ObjectClass::a`仅是作为成员指针的语法被使用。

下面例子说明了如何建立和使用指向数据成员的指针：

```
//: C11:PointerToMemberData.cpp
#include <iostream>
using namespace std;

class Data {
public:
    int a, b, c;
    void print() const {
        cout << "a = " << a << ", b = " << b
            << ", c = " << c << endl;
    }
}
```



```
};

int main() {
    Data d, *dp = &d;
    int Data::*pmInt = &Data::a;
    dp->*pmInt = 47;
    pmInt = &Data::b;
    d.*pmInt = 48;
    pmInt = &Data::c;
    dp->*pmInt = 49;
    dp->print();
} ///:~
```

显然，除了对于一些特例（即需要精确地指向的），这里就显得有些过于难用而无法随处使用。

另外，成员指针是受限制的，它们仅能被指定给在类中的确定的位置。例如，我们不能像使用普通指针那样增加或比较成员指针。

#### 11.4.1 函数

一个类似的练习产生指向成员函数的指针语法。指向函数的指针（参见第3章的最后部分）定义如下：

```
int (*fp)(float);
```

**(\*fp)**的圆括号用来迫使编译器正确判断定义。没有圆括号，这个表达式就是一个返回 **int\***值的函数。

为了定义和使用成员函数的指针，圆括号扮演同样重要的角色。假设在一个结构内有一个函数，通过给普通函数插入类名和作用域运算符就可以定义一个指向成员函数的指针。

```
//: C11:PmemFunDefinition.cpp
class Simple2 {
public:
    int f(float) const { return 1; }
};
int (Simple2::*fp)(float) const;
int (Simple2::*fp2)(float) const = &Simple2::f;
int main() {
    fp = &Simple2::f;
} ///:~
```

从对 **fp2** 定义可以看出，一个成员指针可以在它创建的时候被初始化，或者也可在其他任何时候。不像非成员函数，当获取成员函数的地址时，符号 **&** 不是可选的。但是，可以给出不含参数列表的函数标识符，因为重载方案可以由成员指针的类型所决定。

##### 11.4.1.1 一个例子

在程序运行时，我们可以改变指针所指的内容。因此在运行时就可以通过指针选择或改变我们的行为，这就为程序设计提供了重要的灵活性。成员指针也一样，它允许在运行时选择一个成员。特别的，当类只有公有成员函数（数据成员通常被认为是内部实现的一部分）时，就可以用指针在运行时选择成员函数，下面的例子正是这样：

```
//: C11:PointerToMemberFunction.cpp
```

```

#include <iostream>
using namespace std;

class Widget {
public:
    void f(int) const { cout << "Widget::f()\n"; }
    void g(int) const { cout << "Widget::g()\n"; }
    void h(int) const { cout << "Widget::h()\n"; }
    void i(int) const { cout << "Widget::i()\n"; }
};

int main() {
    Widget w;
    Widget* wp = &w;
    void (Widget::*pmem)(int) const = &Widget::h;
    (w.*pmem)(1);
    (wp->*pmem)(2);
} ///:~

```

当然，期望一般用户创建如此复杂的表达式不是很合乎情理的。如果用户必须直接操作成员指针，那么**typedef**是适合的。为了安排得当，可以使用成员指针作为内部执行机制的一部分。现在回到先前的那个在类中使用成员指针的例子上来。用户所要做的是传递一个数字以选择一个函数<sup>①</sup>。

```

//: C11:PointerToMemberFunction2.cpp
#include <iostream>
using namespace std;

class Widget {
    void f(int) const { cout << "Widget::f()\n"; }
    void g(int) const { cout << "Widget::g()\n"; }
    void h(int) const { cout << "Widget::h()\n"; }
    void i(int) const { cout << "Widget::i()\n"; }
    enum { cnt = 4 };
    void (Widget::*fptr[cnt])(int) const;
public:
    Widget() {
        fptr[0] = &Widget::f; // Full spec required
        fptr[1] = &Widget::g;
        fptr[2] = &Widget::h;
        fptr[3] = &Widget::i;
    }
    void select(int i, int j) {
        if(i < 0 || i >= cnt) return;
        (this->*fptr[i])(j);
    }
    int count() { return cnt; }
};

int main() {
    Widget w;
    for(int i = 0; i < w.count(); i++)
        w.select(i, 47);
} ///:~

```

① 感谢Owen Mortensen提供了本例。



在类接口和`main()`函数里，可以看到，包括函数本身在内的整个实现被隐藏了。代码甚至必须请求对函数的`Count()`。用这个方法，类执行者可以在内部执行时改变函数的数量而不影响使用这个类的代码。

在构造函数中，成员指针的初始化似乎过分指定了。是否可以这样写：

```
fptr[1] = &g;
```

因为名字`g`在成员函数中出现，这是否可以自动地认为在这个类范围内呢？问题是这不符合成员函数的语法，它的语法要求每个人尤其编译器能够判断将要进行什么。相似地，当成员指针被间接引用时，它看起来像这样：

```
(this->*fptr[i])(j);
```

它仍是过分指定的，`this`似乎多余。正如前面所讲的，当它被间接引用时，语法也需要成员指针总是和一个对象绑定在一起。

## 11.5 小结

C++的指针和C中的指针是几乎相等的，这是非常好的。否则，许多C代码在C++中将不会被正确地编译。仅在出现危险赋值的地方，编译器才会产生出错信息。假设确实想这样赋值，编译器的出错可以用简单的（和显式的！）类型转换清除。

C++还从Algol和Pascal中引进引用（*reference*）的概念，引用就像一个能自动被编译器间接引用的常量指针一样。引用占有一个地址，但可以把它看成一个对象。引用是运算符重载语法（第12章的主题）的重点，它也为普通函数按值传递方式传递和返回对象增加了语法上的便利。

拷贝构造函数采用相同类型的已存在对象的引用作为它的参数，它可以被用来从现有的对象创建新对象。当用按值传递方式传递或返回一个对象时，编译器自动调用这个拷贝构造函数。虽然，编译器将自动地创建一个拷贝构造函数，但是，如果认为需要有一个拷贝构造函数，应当自己定义一个，以确保完成正确的操作。如果不想通过按值传递方式传递和返回对象，应该创建一个私有的拷贝构造函数。

指向成员的指针和普通指针一样具有相同的功能：可以在运行时选取特定存储单元（数据或函数）。指向成员的指针只和类成员一起工作，而不是和全局数据或函数。通过使用指向成员的指针，我们的程序设计可以在运行时灵活地改变行为。

## 11.6 练习

部分练习题的答案可以在本书的电子文档“*Annotated Solution Guide for Thinking in C++*”中找到，只需支付很少的费用就可以从<http://www.BruceEckel.com>得到这个电子文档。

- 11-1 把本章开头的“bird & rock”代码段写为C程序（对数据类型使用`structs`），并对它编译，试着用C++的编译器对它进行编译，看看会有什么发生？
- 11-2 把标题为“C++中的引用”的小节的开头部分代码段放入`main()`中，在输出时增加一些说明，以证明引用就相当于被自动间接引用的指针。
- 11-3 写一个程序，在其中尝试（1）创建一个引用，在其创建时没有被初始化。（2）在一个引用被初始化后，改变它的指向，使之指向另一个对象。（3）创建一个NULL引用。

- 11-4 写一个函数，该函数使用指针作为参数，修改指针所指内容，然后用引用返回指针所指的内容。
- 11-5 创建一个包含若干成员函数的类，再用这个类创建一个对象，该对象被练习4中的参数所指向。让这个指针是**const**的和这些成员函数是**const**的，证明仅能在自己的函数内调用**const**成员函数。让函数参数是引用而不是指针。
- 11-6 把标题为“指针引用”小节的开头部分的代码段写成为一段程序。
- 11-7 创建一个函数，使之参数为一个指向指针的指针的引用，要求该函数对其参数进行修改。然后，在**main()**中，调用这个函数。
- 11-8 创建一个函数，使其用**char&**作参数并且修改该参数。在**main()**函数里，打印一个**char**变量，使用这个变量做参数，调用我们设计的函数。然后，再次打印此变量以证明它已被改变。请问这影响了程序的可读性吗？
- 11-9 写一个包含了一个**const**成员函数和一个非**const**成员函数的类，再写三个使用刚创建类的对象作为参数的函数：第一个是通过按值传递方式传递参数，第二个是通过引用方式，第三个是通过**const**引用方式。在这些数的内部，试着调用所创建类的两个成员函数并解释其结果。
- 11-10 （有点挑战性）写一个简单的函数，该函数使用一个**int**作为其参数，增加参数的值并返回它。在**main()**中，调用这个函数。现在观察编译器如何产生汇编代码并且通过汇编描述来追踪，以理解参数是如何被传递和返回的，以及局部变量是如何从栈中索引的。
- 11-11 写一个函数，该函数使用了**char**、**int**、**float**和**double**作为其参数。用编译器产生汇编代码并找出在函数调用之前把参数压入栈的指令。
- 11-12 写一个返回**double**的函数，产生汇编代码并确定该值是如何被返回的。
- 11-13 产生**PassingBigStructures.cpp**的汇编代码，追踪并了解编译器产生代码传送和返回大型结构的方法。
- 11-14 写一个简单的递归函数，该函数减少参数的值，如果参数变为0则返回0，否则调用它本身。产生这个函数的汇编代码，解释编译器创建汇编代码的过程是如何支持递归的。
- 11-15 编写代码用来证明当自己没有创建一个拷贝构造函数时，编译器将自动地生成拷贝构造函数。并证明生成的拷贝构造函数将对基本类型执行位拷贝，而对用户定义的类型执行拷贝构造函数。
- 11-16 创建一个包含拷贝构造函数的类，该类向**cout**说明它被执行。然后创建一个函数，该函数用按值传递方式传递刚创建类的一个对象。再创建一个函数，此函数产生一个刚创建类的局部对象并通过按值传递方式返回它。调用这些函数以证明当通过按值传递方式传递和返回对象时，实际上是调用了拷贝构造函数。
- 11-17 创建一个包含**double\***的类，其构造函数通过调用**new double**来对**double\***进行初始化，并将构造函数的参数中的值赋给结果存储单元。析构函数打印出所指向的值，并把该值设为-1，对存储单元调用**delete**，然后将指针置0。现在创建一个函数，该函数可通过按值传递方式获取刚创建类的一个对象。在**main()**中调用这个函数。看看会有什么问题发生。通过创建一个拷贝构造函数来解决这个问题。
- 11-18 创建一个类，该类中的构造函数就像是一个拷贝构造函数，但它有一个额外的带有默

认值的参数。说明这仍然是作为拷贝构造函数被使用的。

- 11-19 创建一个带有能显示信息的拷贝构造函数的类。再创建第二个类，该类的成员含有一个由第一个类创建的对象，但不创建拷贝构造函数。验证第二个类中自动生成的拷贝构造函数将调用第一个类的拷贝构造函数。
- 11-20 创建一个非常简单的类和一个函数，该函数通过按值传递方式返回所创建类的一个对象。再创建第二个函数，它以一个所创建类的对象的引用为参数。作为第二个函数的参数，调用第一个函数，并说明第二个函数必须在它的参数中使用`const`引用。
- 11-21 创建一个没有拷贝构造函数的简单的类和一个简单的函数，此函数通过按值传递方式接收的参数是所创建类的一个对象。现在通过（仅）对拷贝构造函数增加一个私有声明来改变你的类。请解释当创建的函数被编译时将会发生什么。
- 11-22 本练习会创建一个拷贝构造函数的替代物。创建一个类`X`并声明（但不定义）一个私有类型拷贝构造函数。创建一个公有函数`clone()`以作为一个`const`成员函数，该成员函数返回一个用`new`创建的对像的拷贝。现在写一个函数，使用`const X&`作参数并且复制了一个能被修改的局部拷贝。这种方法的缺点是当你这样做时，必须确保显式地销毁（使用`delete`）被复制的对象。
- 11-23 解释第7章中`Mem.cpp`和`MemTest.cpp`的错误，并解决其问题。
- 11-24 创建一个类，它含有一个`double`类型数据成员和一个打印`double`的`print()`函数。在`main()`中，再分别创建指向所创建类中的数据成员和函数的成员的指针。创建类的一个对象和指向该对象的一个指针，通过指向成员的指针，再使用对象和指向对象的指针，来操纵类的这两种成员。
- 11-25 创建包含一个整型数组的类。能否通过使用指向成员的指针对这个数组进行索引？
- 11-26 通过增加一个重载的成员函数`f()`（你可以决定重载的参数表），修改`PmemFunDefinition.cpp`。再创建一个成员指针，使它指向`f()`的重载版本，然后通过这个指针调用此函数。在这种情况下，重载的结果会如何发生？
- 11-27 根据第3章的`FunctionTable.cpp`，创建包含了一组函数指针的`vector`向量的类，用`add()`和`remove()`成员函数来增加和减少函数指针。再增加一个`run()`函数，该函数可在`vector`中移动，并可调用所有的函数。
- 11-28 修改练习27，使它能够用指向成员函数的指针来完成上述工作。



## 运算符重载

运算符重载 (*operator overloading*) 只是一种“语法上的方便” (*syntactic sugar*), 也就是说它只是另一种函数调用的方式。

其中的不同之处在于函数的参数不是出现在圆括号内, 而是紧贴在一些字符旁边, 这些字符我们一般认为是不可变的运算符。

运算符的使用和普通的函数调用有两点不同。首先语法上是不同的, “调用”运算符时要把运算符放置在参数之间, 有时在参数之后。第二个不同是由编译器决定调用哪一个“函数”。例如, 如果对参数为浮点类型使用运算符“+”, 编译器会“调用”执行浮点类型加法的函数 (这种调用通常是插入内联代码, 或者一段浮点处理器指令)。如果对一个浮点数和一个整数使用运算符“+”, 编译器将“调用”一个特殊的函数, 把 `int` 类型转化为 `float` 类型, 然后再“调用”浮点加法代码。

但在 C++ 中, 可以定义一个处理类的新运算符。这种定义很像一个普通函数的定义, 只是函数的名字由关键字 `operator` 及其后紧跟的运算符组成。差别仅此而已。它像任何其他函数一样也是一个函数, 当编译器遇到适当的模式时, 就会调用这个函数。

### 12.1 两个极端

有些人很容易滥用运算符重载。它确实是一个有趣的工具。但应注意, 它仅仅只是一种语法上的方便, 是另外一种调用函数的方式而已。从这个角度看, 只有在能使涉及类的代码更易写, 尤其是更易读时 (请记住, 读代码的机会比写代码多多了) 才有理由重载运算符。如果不是这样, 就不庸人自扰了。

对于运算符重载, 另外一个常见的反应是恐慌: 突然之间, C 运算符的含义变得不同寻常了。“一切都变了, 所有 C 代码的功能都要改变!” 并非如此。在仅包含内置数据类型的表达式中的所有运算符是不可能被改变的。我们不能重载如下的运算符改变其行为。

```
1 << 4;
```

或者重载运算符使得下面的表达式有意义。

```
1.414 << 2;
```

只有那些包含用户自定义类型的表达式才能有重载的运算符。

### 12.2 语法

定义重载的运算符就像定义函数, 只是该函数的名字是 `operator@`, 这里 `@` 代表了被重载的运算符。函数参数表中参数的个数取决于两个因素:

- 1) 运算符是一元的 (一个参数) 还是二元的 (两个参数)。
- 2) 运算符被定义为全局函数 (对于一元是一个参数, 对于二元是两个参数) 还是成员函数

(对于一元没有参数, 对于二元是一个参数——此时该类的对象用做左侧参数)。

下面的简单类说明了运算符重载的语法:

```
//: C12:OperatorOverloadingSyntax.cpp
#include <iostream>
using namespace std;

class Integer {
    int i;
public:
    Integer(int ii) : i(ii) {}
    const Integer
    operator+(const Integer& rv) const {
        cout << "operator+" << endl;
        return Integer(i + rv.i);
    }
    Integer&
    operator+=(const Integer& rv) {
        cout << "operator+=" << endl;
        i += rv.i;
        return *this;
    }
};

int main() {
    cout << "built-in types:" << endl;
    int i = 1, j = 2, k = 3;
    k += i + j;
    cout << "user-defined types:" << endl;
    Integer ii(1), jj(2), kk(3);
    kk += ii + jj;
} ///:~
```

这两个重载的运算符被定义为内联成员函数, 在它们被调用时会显示信息。对于二元运算符, 惟一的参数是出现在运算符右侧的那个操作数。当一元运算符被定义为成员函数时, 是没有参数的。所调用的成员函数属于运算符左侧的那个对象。

对于非条件运算符 (条件运算符通常返回一个布尔值), 如果两个参数是相同的类型, 总是希望返回相同类型的对象或引用吧 (如果它们不是相同类型, 结果就取决于程序设计者了)。用这种方法可以构造复杂的表达式:

```
kk += ii + jj;
```

**operator +**产生一个新的**Integer** (临时的), 它用做**operator +=**的**rv** (右) 参数。一旦这个临时变量不再需要就会销毁。

### 12.3 可重载的运算符

虽然几乎所有C中的运算符都可以重载, 但运算符重载的使用是相当受限制的。特别是不能使用C中当前没有意义的运算符 (例如用\*\*代表求幂), 不能改变运算符的优先级, 不能改变运算符的参数个数。这样限制有意义, 否则, 所有这些行为产生的运算符只会混淆而不是澄清语意。

下面两个小节给出重载所有“常规”运算符的例子, 重载的形式都是最可能用到的。



### 12.3.1 一元运算符

下面的例子显示了重载所有一元运算符的语法，有全局函数形式（非成员的友元函数）也有成员函数形式。它们将扩充前面给出的类**Integer**并且增加一个新类**byte**。具体运算符的含义取决于使用它们的方式，但在设计特殊操作之前要为未来使用这些类的程序员好好想一想。

这里是所有一元函数的目录：

```
//: C12:OverloadingUnaryOperators.cpp
#include <iostream>
using namespace std;

// Non-member functions:
class Integer {
    long i;
    Integer* This() { return this; }
public:
    Integer(long ll = 0) : i(ll) {}
    // No side effects takes const& argument:
    friend const Integer&
        operator+(const Integer& a);
    friend const Integer
        operator-(const Integer& a);
    friend const Integer
        operator~(const Integer& a);
    friend Integer*
        operator&(Integer& a);
    friend int
        operator!(const Integer& a);
    // Side effects have non-const& argument:
    // Prefix:
    friend const Integer&
        operator++(Integer& a);
    // Postfix:
    friend const Integer
        operator++(Integer& a, int);
    // Prefix:
    friend const Integer&
        operator--(Integer& a);
    // Postfix:
    friend const Integer
        operator--(Integer& a, int);
};

// Global operators:
const Integer& operator+(const Integer& a) {
    cout << "+Integer\n";
    return a; // Unary + has no effect
}
const Integer operator-(const Integer& a) {
    cout << "-Integer\n";
    return Integer(-a.i);
}
const Integer operator~(const Integer& a) {
    cout << "~Integer\n";
    return Integer(~a.i);
}
```



```

}
Integer* operator&(Integer& a) {
    cout << "&Integer\n";
    return a.This(); // &a is recursive!
}
int operator!(const Integer& a) {
    cout << "!Integer\n";
    return !a.i;
}
// Prefix; return incremented value
const Integer& operator++(Integer& a) {
    cout << "++Integer\n";
    a.i++;
    return a;
}
// Postfix; return the value before increment:
const Integer operator++(Integer& a, int) {
    cout << "Integer++\n";
    Integer before(a.i);
    a.i++;
    return before;
}
// Prefix; return decremented value
const Integer& operator--(Integer& a) {
    cout << "--Integer\n";
    a.i--;
    return a;
}
// Postfix; return the value before decrement:
const Integer operator--(Integer& a, int) {
    cout << "Integer--\n";
    Integer before(a.i);
    a.i--;
    return before;
}

// Show that the overloaded operators work:
void f(Integer a) {
    +a;
    -a;
    ~a;
    Integer* ip = &a;
    !a;
    ++a;
    a++;
    --a;
    a--;
}

// Member functions (implicit "this"):
class Byte {
    unsigned char b;
public:
    Byte(unsigned char bb = 0) : b(bb) {}
    // No side effects: const member function:
    const Byte& operator+() const {

```



```

        cout << "+Byte\n";
        return *this;
    }
    const Byte operator-() const {
        cout << "-Byte\n";
        return Byte(-b);
    }
    const Byte operator~() const {
        cout << "~Byte\n";
        return Byte(~b);
    }
    Byte operator!() const {
        cout << "!Byte\n";
        return Byte(!b);
    }
    Byte* operator&() {
        cout << "&Byte\n";
        return this;
    }
    // Side effects: non-const member function:
    const Byte& operator++() { // Prefix
        cout << "++Byte\n";
        b++;
        return *this;
    }
    const Byte operator++(int) { // Postfix
        cout << "Byte++\n";
        Byte before(b);
        b++;
        return before;
    }
    const Byte& operator--() { // Prefix
        cout << "--Byte\n";
        --b;
        return *this;
    }
    const Byte operator--(int) { // Postfix
        cout << "Byte--\n";
        Byte before(b);
        --b;
        return before;
    }
};

void g(Byte b) {
    +b;
    -b;
    ~b;
    Byte* bp = &b;
    !b;
    ++b;
    b++;
    --b;
    b--;
}

```



```
int main() {
    Integer a;
    f(a);
    Byte b;
    g(b);
} ///:~
```

函数是根据其参数传递的方法分组的。如何传递和返回参数的指导方针到后面再讲。上面的形式（和下一小节的形式）是典型的使用形式，所以在你自己重载运算符时可以从这些范式开始。

#### 12.3.1.1 自增和自减

重载的++和--运算符有点让人进退维谷，因为我们总是希望能根据它们出现在所作用的对象的前面（前缀）还是后面（后缀）来调用不同的函数。解决方法很简单，但有些人一开始会觉得容易混淆。例如当编译器看到++a（先自增）时，它就调用operator++(a)；但当编译器看到a++时，它就调用operator++(a,int)。即编译器通过调用不同的重载函数区别这两种形式。在成员函数版本的OverloadingUnaryOperators.cpp中，如果编译器看到++b，它就产生一个对B::operator++()的调用；如果编译器看到b++，它就产生一个对B::operator++(int)的调用。

用户所见到的是对前缀和后缀版本调用不同的函数。然而，实质上这两个函数调用有着不同的标记，所以它们指向两个不同的函数体。编译器为int参数传递一个哑元常量值（因为这个值永远不被使用，所以它永远不会给出一个标识符）用来为后缀版产生不同的标记。

#### 12.3.2 二元运算符

下面的清单为二元运算符重复了OverloadingUnaryOperators.cpp，于是就有了所有可重载运算符的例子。全局版本和成员函数版本都在里面。

```
///: C12:Integer.h
// Non-member overloaded operators
#ifndef INTEGER_H
#define INTEGER_H
#include <iostream>

// Non-member functions:
class Integer {
    long i;
public:
    Integer(long ll = 0) : i(ll) {}
    // Operators that create new, modified value:
    friend const Integer
        operator+(const Integer& left,
                  const Integer& right);
    friend const Integer
        operator-(const Integer& left,
                  const Integer& right);
    friend const Integer
        operator*(const Integer& left,
                  const Integer& right);
    friend const Integer
        operator/(const Integer& left,
```



```

        const Integer& right);
friend const Integer
    operator%(const Integer& left,
               const Integer& right);
friend const Integer
    operator^(const Integer& left,
               const Integer& right);
friend const Integer
    operator&(const Integer& left,
               const Integer& right);
friend const Integer
    operator|(const Integer& left,
               const Integer& right);
friend const Integer
    operator<<(const Integer& left,
               const Integer& right);
friend const Integer
    operator>>(const Integer& left,
               const Integer& right);
// Assignments modify & return lvalue:
friend Integer&
    operator+=(Integer& left,
               const Integer& right);
friend Integer&
    operator-=(Integer& left,
               const Integer& right);
friend Integer&
    operator*=(Integer& left,
               const Integer& right);
friend Integer&
    operator/=(Integer& left,
               const Integer& right);
friend Integer&
    operator%=(Integer& left,
               const Integer& right);
friend Integer&
    operator^=(Integer& left,
               const Integer& right);
friend Integer&
    operator&=(Integer& left,
               const Integer& right);
friend Integer&
    operator|=(Integer& left,
               const Integer& right);
friend Integer&
    operator>>=(Integer& left,
               const Integer& right);
friend Integer&
    operator<<=(Integer& left,
               const Integer& right);
// Conditional operators return true/false:
friend int
    operator==(const Integer& left,
               const Integer& right);
friend int
    operator!=(const Integer& left,

```



```

        const Integer& right);
friend int
    operator<(const Integer& left,
              const Integer& right);
friend int
    operator>(const Integer& left,
              const Integer& right);
friend int
    operator<=(const Integer& left,
               const Integer& right);
friend int
    operator>=(const Integer& left,
               const Integer& right);
friend int
    operator&&(const Integer& left,
               const Integer& right);
friend int
    operator||(const Integer& left,
                const Integer& right);
// Write the contents to an ostream:
void print(std::ostream& os) const { os << i; }
};
#endif // INTEGER_H ///:~

//: C12:Integer.cpp {0}
// Implementation of overloaded operators
#include "Integer.h"
#include "../require.h"

const Integer
    operator+(const Integer& left,
              const Integer& right) {
    return Integer(left.i + right.i);
}
const Integer
    operator-(const Integer& left,
              const Integer& right) {
    return Integer(left.i - right.i);
}
const Integer
    operator*(const Integer& left,
              const Integer& right) {
    return Integer(left.i * right.i);
}
const Integer
    operator/(const Integer& left,
              const Integer& right) {
    require(right.i != 0, "divide by zero");
    return Integer(left.i / right.i);
}
const Integer
    operator%(const Integer& left,
              const Integer& right) {
    require(right.i != 0, "modulo by zero");
    return Integer(left.i % right.i);
}

```



```

const Integer
    operator^(const Integer& left,
               const Integer& right) {
    return Integer(left.i ^ right.i);
}
const Integer
    operator&(const Integer& left,
               const Integer& right) {
    return Integer(left.i & right.i);
}
const Integer
    operator|(const Integer& left,
               const Integer& right) {
    return Integer(left.i | right.i);
}
const Integer
    operator<<(const Integer& left,
                const Integer& right) {
    return Integer(left.i << right.i);
}
const Integer
    operator>>(const Integer& left,
                const Integer& right) {
    return Integer(left.i >> right.i);
}
// Assignments modify & return lvalue:
Integer& operator+=(Integer& left,
                     const Integer& right) {
    if(&left == &right) {/* self-assignment */}
    left.i += right.i;
    return left;
}
Integer& operator-=(Integer& left,
                     const Integer& right) {
    if(&left == &right) {/* self-assignment */}
    left.i -= right.i;
    return left;
}
Integer& operator*=(Integer& left,
                     const Integer& right) {
    if(&left == &right) {/* self-assignment */}
    left.i *= right.i;
    return left;
}
Integer& operator/=(Integer& left,
                     const Integer& right) {
    require(right.i != 0, "divide by zero");
    if(&left == &right) {/* self-assignment */}
    left.i /= right.i;
    return left;
}
Integer& operator%=(Integer& left,
                     const Integer& right) {
    require(right.i != 0, "modulo by zero");
    if(&left == &right) {/* self-assignment */}
    left.i %= right.i;

```



```

    return left;
}
Integer& operator^=(Integer& left,
                    const Integer& right) {
    if(&left == &right) { /* self-assignment */
        left.i ^= right.i;
    }
    return left;
}
Integer& operator&=(Integer& left,
                    const Integer& right) {
    if(&left == &right) { /* self-assignment */
        left.i &= right.i;
    }
    return left;
}
Integer& operator|=(Integer& left,
                    const Integer& right) {
    if(&left == &right) { /* self-assignment */
        left.i |= right.i;
    }
    return left;
}
Integer& operator>>=(Integer& left,
                    const Integer& right) {
    if(&left == &right) { /* self-assignment */
        left.i >>= right.i;
    }
    return left;
}
Integer& operator<<=(Integer& left,
                    const Integer& right) {
    if(&left == &right) { /* self-assignment */
        left.i <<= right.i;
    }
    return left;
}
// Conditional operators return true/false:
int operator==(const Integer& left,
               const Integer& right) {
    return left.i == right.i;
}
int operator!=(const Integer& left,
               const Integer& right) {
    return left.i != right.i;
}
int operator<(const Integer& left,
              const Integer& right) {
    return left.i < right.i;
}
int operator>(const Integer& left,
              const Integer& right) {
    return left.i > right.i;
}
int operator<=(const Integer& left,
               const Integer& right) {
    return left.i <= right.i;
}
int operator>=(const Integer& left,
               const Integer& right) {
    return left.i >= right.i;
}

```





```

    }
    int operator&&(const Integer& left,
                  const Integer& right) {
        return left.i && right.i;
    }
    int operator|| (const Integer& left,
                  const Integer& right) {
        return left.i || right.i;
    } ///:~

//: C12:IntegerTest.cpp
//{L} Integer
#include "Integer.h"
#include <fstream>
using namespace std;
ofstream out("IntegerTest.out");

void h(Integer& c1, Integer& c2) {
    // A complex expression:
    c1 += c1 * c2 + c2 % c1;
    #define TRY(OP) \
        out << "c1 = "; c1.print(out); \
        out << ", c2 = "; c2.print(out); \
        out << "; c1 " #OP " c2 produces "; \
        (c1 OP c2).print(out); \
        out << endl;
    TRY(+) TRY(-) TRY(*) TRY(/)
    TRY(%) TRY(^) TRY(&) TRY(||)
    TRY(<<) TRY(>>) TRY(+=) TRY(-=)
    TRY(*=) TRY(/=) TRY(%=) TRY(^=)
    TRY(&=) TRY(|=) TRY(>>=) TRY(<<=)
    // Conditionals:
    #define TRYC(OP) \
        out << "c1 = "; c1.print(out); \
        out << ", c2 = "; c2.print(out); \
        out << "; c1 " #OP " c2 produces "; \
        out << (c1 OP c2); \
        out << endl;
    TRYC(<) TRYC(>) TRYC(==) TRYC(!=) TRYC(<=)
    TRYC(>=) TRYC(&&) TRYC(||)
}

int main() {
    cout << "friend functions" << endl;
    Integer c1(47), c2(9);
    h(c1, c2);
} ///:~

//: C12:Byte.h
// Member overloaded operators
#ifndef BYTE_H
#define BYTE_H
#include "../require.h"
#include <iostream>
// Member functions (implicit "this"):
class Byte {
    unsigned char b;

```



```

public:
    Byte(unsigned char bb = 0) : b(bb) {}
    // No side effects: const member function:
    const Byte
        operator+(const Byte& right) const {
            return Byte(b + right.b);
        }
    const Byte
        operator-(const Byte& right) const {
            return Byte(b - right.b);
        }
    const Byte
        operator*(const Byte& right) const {
            return Byte(b * right.b);
        }
    const Byte
        operator/(const Byte& right) const {
            require(right.b != 0, "divide by zero");
            return Byte(b / right.b);
        }
    const Byte
        operator%(const Byte& right) const {
            require(right.b != 0, "modulo by zero");
            return Byte(b % right.b);
        }
    const Byte
        operator^(const Byte& right) const {
            return Byte(b ^ right.b);
        }
    const Byte
        operator&(const Byte& right) const {
            return Byte(b & right.b);
        }
    const Byte
        operator|(const Byte& right) const {
            return Byte(b | right.b);
        }
    const Byte
        operator<<(const Byte& right) const {
            return Byte(b << right.b);
        }
    const Byte
        operator>>(const Byte& right) const {
            return Byte(b >> right.b);
        }
    // Assignments modify & return lvalue.
    // operator= can only be a member function:
    Byte& operator=(const Byte& right) {
        // Handle self-assignment:
        if(this == &right) return *this;
        b = right.b;
        return *this;
    }
    Byte& operator+=(const Byte& right) {
        if(this == &right) { /* self-assignment */
            b += right.b;

```



```

    return *this;
}
Byte& operator==(const Byte& right) {
    if(this == &right) {/* self-assignment */}
    b -= right.b;
    return *this;
}
Byte& operator*=(const Byte& right) {
    if(this == &right) {/* self-assignment */}
    b *= right.b;
    return *this;
}
Byte& operator/=(const Byte& right) {
    require(right.b != 0, "divide by zero");
    if(this == &right) {/* self-assignment */}
    b /= right.b;
    return *this;
}
Byte& operator%=(const Byte& right) {
    require(right.b != 0, "modulo by zero");
    if(this == &right) {/* self-assignment */}
    b %= right.b;
    return *this;
}
Byte& operator^=(const Byte& right) {
    if(this == &right) {/* self-assignment */}
    b ^= right.b;
    return *this;
}
Byte& operator&=(const Byte& right) {
    if(this == &right) {/* self-assignment */}
    b &= right.b;
    return *this;
}
Byte& operator|=(const Byte& right) {
    if(this == &right) {/* self-assignment */}
    b |= right.b;
    return *this;
}
Byte& operator>>=(const Byte& right) {
    if(this == &right) {/* self-assignment */}
    b >>= right.b;
    return *this;
}
Byte& operator<<=(const Byte& right) {
    if(this == &right) {/* self-assignment */}
    b <<= right.b;
    return *this;
}
// Conditional operators return true/false:
int operator==(const Byte& right) const {
    return b == right.b;
}
int operator!=(const Byte& right) const {
    return b != right.b;
}

```



```

int operator<(const Byte& right) const {
    return b < right.b;
}
int operator>(const Byte& right) const {
    return b > right.b;
}
int operator<=(const Byte& right) const {
    return b <= right.b;
}
int operator>=(const Byte& right) const {
    return b >= right.b;
}
int operator&&(const Byte& right) const {
    return b && right.b;
}
int operator|| (const Byte& right) const {
    return b || right.b;
}
// Write the contents to an ostream:
void print(std::ostream& os) const {
    os << "0x" << std::hex << int(b) << std::dec;
}
};
#endif // BYTE_H ///:~

//: C12:ByteTest.cpp
#include "Byte.h"
#include <fstream>
using namespace std;
ofstream out("ByteTest.out");
void k(Byte& b1, Byte& b2) {
    b1 = b1 * b2 + b2 % b1;

#define TRY2(OP) \
    out << "b1 = "; b1.print(out); \
    out << ", b2 = "; b2.print(out); \
    out << "; b1 " #OP " b2 produces "; \
    (b1 OP b2).print(out); \
    out << endl;

    b1 = 9; b2 = 47;
    TRY2(+) TRY2(-) TRY2(*) TRY2(/)
    TRY2(%) TRY2(^) TRY2(&) TRY2(|)
    TRY2(<<) TRY2(>>) TRY2(+=) TRY2(-=)
    TRY2(*=) TRY2(/=) TRY2(%=) TRY2(^=)
    TRY2(&=) TRY2(|=) TRY2(>>=) TRY2(<<=)
    TRY2(=) // Assignment operator

    // Conditionals:
#define TRYC2(OP) \
    out << "b1 = "; b1.print(out); \
    out << ", b2 = "; b2.print(out); \
    out << "; b1 " #OP " b2 produces "; \
    out << (b1 OP b2); \
    out << endl;

```



```

    b1 = 9; b2 = 47;
    TRYC2(<) TRYC2(>) TRYC2(==) TRYC2(!=) TRYC2(<=)
    TRYC2(>=) TRYC2(&&) TRYC2(||)

    // Chained assignment:
    Byte b3 = 92;
    b1 = b2 = b3;
}

int main() {
    out << "member functions:" << endl;
    Byte b1(47), b2(9);
    k(b1, b2);
} ///:~

```

可以看到`operator=`只允许作为成员函数。这将在后面解释。

请注意在运算符重载中所有赋值运算符都有代码检测自赋值(self-assignment), 这是总原则。在某些情况下, 这是不需要的。例如, 对于`operator+=`, 我们总是习惯写`A+=A`, 让A与自己相加。检测自赋值最重要的地方是`operator=`, 因为复杂的对象可能因为它而发生灾难性的结果(在一些情况下这不会有问题, 但不管怎么说, 在写`operator=`时, 应该小心一些)。

在前两个例子中重载的运算符处理的是单一类型。也可以重载运算符处理混合类型, 所以可以“将苹果与橙子相加”。然而, 在开始进行运算符重载之前, 应该看一下本章后面有关自动类型转换的一节。在适当的地方使用类型转换可以减少许多运算符重载。

### 12.3.3 参数和返回值

在`OverloadingUnaryOperators.cpp`、`Integer.h`和`Byte.h`例子中可以见到各种不同的参数传递和返回方法, 乍一看让人有些摸不着头脑。虽然可以用任何需要的方式传递和返回参数, 但在这些例子中所用的方式却不是随便选的。它们遵守一种合乎逻辑的模式, 我们在大部分情况下都应选择这种模式:

- 1) 对于任何函数参数, 如果仅需要从参数中读而不改变它, 默认地应当作为`const`引用来传递它。普通算术运算符(像“+”和“-”等)和布尔运算符不会改变参数, 所以以`const`引用传递是主要的使用方式。当函数是一个类成员的时候, 就转换为`const`成员函数。只有会改变左侧参数的运算符赋值(operator-assignment)(如`+=`)和`operator=`, 左侧参数不是常量, 但因为参数将被改变, 所以参数仍然按地址传递。
- 2) 返回值的类型取决于运算符的具体含义(我们可以对参数和返回值做任何想做的事)。如果使用该运算符的结果是产生一个新值, 就需要产生一个作为返回值的新对象。例如, `Integer::operator+`必须生成一个操作数之和的`Integer`对象。这个对象作为一个常量通过传值方式返回, 所以作为一个左值结果不会被改变。
- 3) 所有赋值运算符均改变左值。为了使赋值结果能用于链式表达式(如`a=b=c`), 应该能够返回一个刚刚改变了的左值的引用。但这个引用应该是常量还是非常量呢? 虽然我们是从左向右读表达式`a=b=c`, 但编译器是从右向左分析这个表达式, 所以并非一定要返回一个非常量值来支持链式赋值。然而人们有时希望能够对刚刚赋值的对象进行运算, 例如`(a=b).func()`, 这是b赋值给a后调用`func()`。因此所有赋值运算

符的返回值对于左值应该是非常量引用。

- 4) 对于逻辑运算符，人们希望至少得到一个`int`返回值，最好是`bool`返回值（在大多数编译器支持C++内置`bool`类型之前开发的库函数使用`int`或者用`typedef`产生的等价类型）。

因为有前缀和后缀版本，所以自增和自减运算符出现了两难局面。由于两个版本都改变了对象，所以这个对象不能作为常量类型。在对象被改变后，前缀版本返回其值，我们希望返回改变后的对象。这样，用前缀版本只需作为一个引用返回`*this`。因为后缀版本返回改变之前的值，所以必须创建一个代表这个值的独立对象并返回它。因此，如果想保持本意，对于后缀必须通过传值方式返回。（注意，我们经常会发现自增和自减运算符返回一个`int`值或`bool`值，表示诸如是否在列表上移动的对象到达了列表尾部这样的情况）。现在的问题是：它们应该按常量还是按非常量返回？如果允许对象被改变，而有的人写了表达式`(++a).func()`，那么`func()`作用在`a`上。但对于表达式`(a++).func()`，`func()`作用在通过后缀`operator++`返回的临时对象上。临时对象自动定为常量，所以这一操作会被编译器阻止。但为了一致性，两者都是常量更有意义，这里就是如此。我们可以选择让前缀版本是非常量的，而后缀版本是常量的。因为想给自增和自减运算符赋予各种含义，所以它们需要就事论事考虑。

#### 12.3.3.1 作为常量通过传值方式返回

作为常量通过传值方式返回，开始看起来有些微妙，所以值得多加解释。现在考虑二元运算符`+`。假设在表达式`f(a+b)`中使用它，`a+b`的结果变为一个临时对象，这个对象用于对`f()`的调用中。因为它是临时的，自动被定为常量，所以无论是否使返回值为常量都没有影响。

然而，也可能发送一个消息给`a+b`的返回值而不是仅传递给一个函数。例如，可以写表达式`(a+b).g()`，其中`g()`是`Integer`的成员函数。这里，通过设返回值为常量，规定了对于返回值只有常量成员函数才可以被调用。用常量是恰当的，这是因为这样可以使我们不用在对象中存储可能有价值的信息，而该信息很可能是会被丢失的。

#### 12.3.3.2 返回值优化

通过传值方式返回要创建新对象时，应注意使用的形式。例如在`operator+`：

```
return Integer(left.i + right.i);
```

乍看起来这像是一个“对一个构造函数的调用”，其实并非如此。这是临时对象语法，它是在说：“创建一个临时`Integer`对象并返回它”。据此我们可能认为如果创建一个有名字的局部对象并返回它结果将会是一样的。其实不然。如果如下编写，

```
Integer tmp(left.i + right.i);
return tmp;
```

将发生三件事。首先，创建`tmp`对象，其中包括构造函数的调用。然后，拷贝构造函数把`tmp`拷贝到外部返回值的存储单元里。最后，当`tmp`在作用域的结尾时调用析构函数。

相反，“返回临时对象”的方式是完全不同的。当编译器看到我们这样做时，它明白对创建的对象没有其他需求，只是返回它，所以编译器直接地把这个对象创建在外部返回值的内存单元。因为不是真正创建一个局部对象，所以仅需要一个普通构造函数调用（不需要拷贝构造函数），且不会调用析构函数。这种方法不需要什么花费，因此效率是非常高的，但程序员要理解这些。这种方式常被称作返回值优化（*return value optimization*）。

### 12.3.4 不常用的运算符

还有一些运算符的重载语法有一点不同。

下标运算符`operator[]`，必须是成员函数并且它只接受一个参数。因为它所作用的对象应该像数组一样操作，可以经常从这个运算符返回一个引用，所以它可以被很方便地用于等号左侧。这个运算符经常被重载，可以在本书其他部分看到相关的例子。

运算符`new`和`delete`用于控制动态存储分配并能按许多种不同的方法进行重载，这将在第13章中讨论。

#### 12.3.4.1 `operator,`

当逗号出现在一个对象左右，而该对象的类型是逗号定义所支持的类型时，将调用逗号运算符。然而，“`operator,`”调用的目标不是函数参数表，而是被逗号分隔开的、没有被括号括起来的对象。除了使语言保持一致性外，这个运算符似乎没有许多实际用途。下面的例子说明了当逗号出现在对象前面以及后面时，逗号函数调用的方式：

```
//: C12:OverloadingOperatorComma.cpp
#include <iostream>
using namespace std;

class After {
public:
    const After& operator,(const After&) const {
        cout << "After::operator,()" << endl;
        return *this;
    }
};

class Before {};

Before& operator,(int, Before& b) {
    cout << "Before::operator,()" << endl;
    return b;
}

int main() {
    After a, b;
    a, b; // Operator comma called

    Before c;
    1, c; // Operator comma called
} ///:~
```

全局函数允许逗号放在被讨论的对象的前面。这里的用法既晦涩又可疑。虽然还可以再把一个逗号分隔的参数表当做更加复杂的表达式的一部分，但这太灵活了，大多数情况下不能使用。

#### 12.3.4.2 `operator->`

当希望一个对象表现得像一个指针时，通常就要用到`operator->`。由于这样一个对象比一般的指针有着更多与生俱来的灵巧性，于是常被称作灵巧指针（*smart pointer*）。如果想用类包装一个指针以使指针安全，或是在迭代器（*iterator*）普通的用法中，这样做会特别有用。迭代器是一个对象，这个对象可以作用于其他对象的容器或集合上，每次选择它们中的一个，

而不用提供对容器的直接访问。(在类函数里经常发现容器和迭代器,例如本书第2卷中描述的标准C++库。)

指针间接引用运算符一定是一个成员函数。它有着额外的、非典型的限制:它必须返回一个对象(或对象的引用),该对象也有一个指针间接引用运算符;或者必须返回一个指针,被用于选择指针间接引用运算符箭头所指向的内容。下面是一个简单的例子:

```

//: C12:SmartPointer.cpp
#include <iostream>
#include <vector>
#include "../require.h"
using namespace std;

class Obj {
    static int i, j;
public:
    void f() const { cout << i++ << endl; }
    void g() const { cout << j++ << endl; }
};

// Static member definitions:
int Obj::i = 47;
int Obj::j = 11;

// Container:
class ObjContainer {
    vector<Obj*> a;
public:
    void add(Obj* obj) { a.push_back(obj); }
    friend class SmartPointer;
};

class SmartPointer {
    ObjContainer& oc;
    int index;
public:
    SmartPointer(ObjContainer& objc) : oc(objc) {
        index = 0;
    }
    // Return value indicates end of list:
    bool operator++() { // Prefix
        if(index >= oc.a.size()) return false;
        if(oc.a[++index] == 0) return false;
        return true;
    }
    bool operator++(int) { // Postfix
        return operator++(); // Use prefix version
    }
    Obj* operator->() const {
        require(oc.a[index] != 0, "Zero value "
            "returned by SmartPointer::operator->()");
        return oc.a[index];
    }
};

```





```
int main() {
    const int sz = 10;
    Obj o[sz];
    ObjContainer oc;
    for(int i = 0; i < sz; i++)
        oc.add(&o[i]); // Fill it up
    SmartPointer sp(oc); // Create an iterator
    do {
        sp->f(); // Pointer dereference operator call
        sp->g();
    } while(sp++);
} ///:~
```

类**Obj**定义了程序中使用的一些对象。函数**f()**和**g()**用静态数据成员打印令人感兴趣的值。使用**ObjContainer**的函数**add()**将指向这些对象的指针存储在类型为**ObjContainer**的容器中。**ObjContainer**看起来像一个指针数组，但却没有办法取回这些指针。然而，类**SmartPointer**被声明为友元类，所以它允许进入这个容器内。类**SmartPointer**看起来像一个聪明的指针——可以使用运算符**++**向前移动它（也可以定义一个**operator--**），它不会超出它所指向的容器的范围，它可以返回它指向的内容（通过这个指针间接引用运算符）。注意，不像一个基本指针，**SmartPointer**是和所创建的容器的配套使用的，不存在一个具有“通用目的”的灵巧指针。我们将在本书最后一章和第2卷中了解更多被称为“迭代器”的灵巧指针的内容。

在**main()**中，一旦**Obj**对象装入容器**oc**，一个**SmartPointer**类的**SP**就创建了。灵巧指针按下面的表达式进行调用：

```
sp->f(); // Smart pointer calls
sp->g();
```

这里，尽管**sp**实际上并没有成员函数**f()**和**g()**，但指针间接引用运算符自动地为用**SmartPointer::operator-->**返回的**Obj\***调用那些函数。编译器进行所有检查以保证函数调用正确。

虽然，指针间接运算符的底层机制比其他运算符复杂一些，但目的是一样的——为类的用户提供更为方便的语法。

#### 12.3.4.3 嵌入的迭代器

更常见的是，“灵巧指针”和“迭代器”类嵌入它所服务的类中。前面的例子可按如下重写，以在**ObjContainer**中嵌入**SmartPointer**。

```
///C12:NestedSmartPointer.cpp
#include <iostream>
#include <vector>
#include "../require.h"
using namespace std;

class Obj {
    static int i, j;
public:
    void f() { cout << i++ << endl; }
    void g() { cout << j++ << endl; }
};

// Static member definitions:
int Obj::i = 47;
```



```

int Obj::j = 11;

// Container:
class ObjContainer {
    vector<Obj*> a;
public:
    void add(Obj* obj) { a.push_back(obj); }
    class SmartPointer;
    friend SmartPointer;
    class SmartPointer {
        ObjContainer& oc;
        unsigned int index;
    public:
        SmartPointer(ObjContainer& objc) : oc(objc) {
            index = 0;
        }
        // Return value indicates end of list:
        bool operator++() { // Prefix
            if(index >= oc.a.size()) return false;
            if(oc.a[++index] == 0) return false;
            return true;
        }
        bool operator++(int) { // Postfix
            return operator++(); // Use prefix version
        }
        Obj* operator->() const {
            require(oc.a[index] != 0, "Zero value "
                "returned by SmartPointer::operator->()");
            return oc.a[index];
        }
    };
    // Function to produce a smart pointer that
    // points to the beginning of the ObjContainer:
    SmartPointer begin() {
        return SmartPointer(*this);
    }
};

int main() {
    const int sz = 10;
    Obj o[sz];
    ObjContainer oc;
    for(int i = 0; i < sz; i++)
        oc.add(&o[i]); // Fill it up
    ObjContainer::SmartPointer sp = oc.begin();
    do {
        sp->f(); // Pointer dereference operator call
        sp->g();
    } while(++sp);
} ///:~

```

除了实际上嵌入了类中，另有两点不同之处。首先是在类的声明中说明它是一个友元类。

```

class SmartPointer;
friend SmartPointer;

```

编译器首先在被告知类是友元的之前，必须知道该类是存在的。

第二个不同之处是在**ObjContainer**的成员函数**begin()**中，**begin()**产生一个指向**ObjContainer**序列开头的**SmartPointer**。虽然实际上仅是方便了，但由于它遵循了在标准C++库中使用的部分形式，所以还是值得的。

#### 12.3.4.4 operator->\*

**operator->\***是一个二元运算符，其行为与所有其他二元运算符类似。它是专为模仿前一章介绍的内建数据类型的成员指针行为而提供的。

与**operator->\***一样，指向成员的指针间接引用运算符通常同某种代表“灵巧指针”的对象一起使用。这里的例子将简单些以便于理解。在定义**operator->\***时要注意它必须返回一个对象，对于这个对象，可以用正在调用的成员函数为参数调用**operator()**。

**operator()**的函数调用必须是成员函数，它是惟一的允许在它里面有任意个参数的函数。这使得对象看起来像一个真正的函数。虽然可以定义一些重载的带不同参数的**operator()**函数，但这常被用于仅有一个单一操作数或至少是一个特别优先的类型。在第2卷中，可以看到标准C++库使用函数调用运算符以创建“函数对象”。

要想创建一个**operator->\***，必须首先创建带有**operator()**类，这是**operator->\***将返回对象的类。该类必须获取一些必要的信息，以使当**operator()**被调用时，指向成员的指针可以对对象进行间接引用。在下面的例子中，**FunctionObject**的构造函数得到并储存指向对象的指针和指向成员函数的指针，然后**operator()**使用这些指针进行实际指向成员的指针的调用。

```
//: C12:PointerToMemberOperator.cpp
#include <iostream>
using namespace std;

class Dog {
public:
    int run(int i) const {
        cout << "run\n";
        return i;
    }
    int eat(int i) const {
        cout << "eat\n";
        return i;
    }
    int sleep(int i) const {
        cout << "ZZZ\n";
        return i;
    }
    typedef int (Dog::*PMF)(int) const;
    // operator->* must return an object
    // that has an operator():
    class FunctionObject {
        Dog* ptr;
        PMF pmem;
    public:
        // Save the object pointer and member pointer
        FunctionObject(Dog* wp, PMF pmf)
            : ptr(wp), pmem(pmf) {
            cout << "FunctionObject constructor\n";
        }
        // Make the call using the object pointer
```



```

// and member pointer
int operator()(int i) const {
    cout << "FunctionObject::operator()\n";
    return (ptr->*pmem)(i); // Make the call
}
};
FunctionObject operator->*(PMF pmf) {
    cout << "operator->*" << endl;
    return FunctionObject(this, pmf);
}
};

int main() {
    Dog w;
    Dog::PMF pmf = &Dog::run;
    cout << (w->*pmf)(1) << endl;
    pmf = &Dog::sleep;
    cout << (w->*pmf)(2) << endl;
    pmf = &Dog::eat;
    cout << (w->*pmf)(3) << endl;
} ///:~

```

**Dog**有三个成员函数，它们的参数和返回类型都是**int**。**PMF**是一个**typedef**，用于简化定义一个指向**Dog**成员函数的指向成员的指针。

**operator->\***创建并返回一个**FunctionObject**对象。注意**operator->\***既知道指向成员的指针所调用的对象（**this**），又知道这个指向成员的指针，并把它们传递给存储这些值的**FunctionObject**构造函数。当**operator->\***被调用时，编译器立刻转而对**operator->\***返回的值调用**operator()**，把已经给**operator->\***的参数传递进去。**FunctionObject::operator()**得到参数，然后使用存储的对象指针和指向成员的指针间接引用“真实的”指向成员的指针。

注意，正如**operator->**，这里操作的内容正插入到调用**operator->\***的中间。如果需要的话，这允许我们执行某些额外的操作。

此处执行的**operator->\***机制仅作用于参数和返回值是**int**的成员函数。这是一个局限，但是，如果试着为每一个不同的可能性进行重载，这就像是一个禁止的行为。幸运的是，C++的**template**机制（在本书的最后一章和第2卷中讲述）被设计用来处理这个问题。

### 12.3.5 不能重载的运算符

在可用的运算符集合里存在一些不能重载的运算符。这样限制的通常原因是出于对安全的考虑：如果这些运算符也可以被重载，将会造成危害或破坏安全机制，使得事情变得困难或混淆现有的习惯。

- 1) 成员选择**operator.**。点在类中对任何成员都有一定的意义。但如果允许它重载，就不能用普通的方法访问成员，只能用指针和指针**operator->**访问。
- 2) 成员指针间接引用**operator.\***，因为与**operator.**同样的原因而不能重载。
- 3) 没有求幂运算符。通常的选择是来自Fortran语言的**operator\*\***，但这出现了难以分析的问题。C没有求幂运算符，C++似乎也不需要，因为这可以通过函数调用来实现。求幂运算符增加了使用的方便，但没有增加新的语言功能，反而为编译器增加了复杂性。
- 4) 不存在用户定义的运算符，即不能编写目前运算符集合中没有的运算符。不能这样做

的部分原因是难以决定其优先级，另一部分原因是没有必要增加麻烦。

5) 不能改变优先级规则。否则人们很难记住它们。

## 12.4 非成员运算符

在前面的一些例子里，运算符可能是成员运算符或非成员运算符，这似乎没有多大差异。这样就会出现一个问题：“应该选择哪一种？”总的来说，如果没有什么差异，它们应该是成员运算符。这样做强调了运算符和类的联合。当左侧操作数是当前类的对象时，运算符会工作得很好。

但有时左侧运算符是别的类的对象。这种情况通常出现在为输入输出流重载operator<<和>>时。因为输入输出流是一个基本C++库，我们将有可能想为定义的大部分类重载运算符，所以这个过程是值得记住的：

```

//: C12:IostreamOperatorOverloading.cpp
// Example of non-member overloaded operators
#include "../require.h"
#include <iostream>
#include <sstream> // "String streams"
#include <cstring>
using namespace std;

class IntArray {
    enum { sz = 5 };
    int i[sz];
public:
    IntArray() { memset(i, 0, sz* sizeof(*i)); }
    int& operator[](int x) {
        require(x >= 0 && x < sz,
            "IntArray::operator[] out of range");
        return i[x];
    }
    friend ostream&
        operator<<(ostream& os, const IntArray& ia);
    friend istream&
        operator>>(istream& is, IntArray& ia);
};

ostream&
operator<<(ostream& os, const IntArray& ia) {
    for(int j = 0; j < ia.sz; j++) {
        os << ia.i[j];
        if(j != ia.sz - 1)
            os << ", ";
    }
    os << endl;
    return os;
}

istream& operator>>(istream& is, IntArray& ia){
    for(int j = 0; j < ia.sz; j++)
        is >> ia.i[j];
    return is;
}

```



```
int main() {
    stringstream input("47 34 56 92 103");
    IntArray I;
    input >> I;
    I[4] = -1; // Use overloaded operator[]
    cout << I;
} ///:~
```

这个类还包含重载 `operator[]`，这个运算符在数组里返回了一个合法值的引用。因为一个引用被返回，所以下面的表达式：

```
I[4] = -1;
```

看起来不仅比使用指针更规范些，而且它也达到了预期的效果。

被重载的移位运算符通过引用方式传递和返回，所以运算将影响外部对象。在函数定义中，表达式如

```
os << ia.i[j];
```

会使现有的重载运算符函数被调用（即那些定义在 `<iostream>` 中的）。在这种情况下，被调用的函数是 `ostream& operator<<(ostream&,int)`，这是因为 `ia.i[j]` 是一个 `int` 值。

一旦所有的动作在 `istream` 或 `ostream` 上完成，它将被返回，因此它可被用于更复杂的表达式。

在 `main()` 中，`istream` 的一种新类型被使用：`stringstream`（在 `<sstream>` 中声明）。该类包含一个 `string`（正如此处显示的，它可以由一个 `char` 数组创建）并且把它转化为一个 `istream`。在上面的例子中，这意味着在不打开一个文件或在命令行中键入数据的情况下，移位运算符可以被测试。

这个例子使用的是插入符和提取符的标准形式。如果想为自己的类创建一个集合，可以拷贝这个函数署名并返回以上的类型，且遵从该函数体的形式。

#### 12.4.1 基本方针

Murray<sup>①</sup>为在成员和非成员之间的选择提出了如下的方针：

运算符	建议使用
所有的一元运算符	成员
<code>= () [] -&gt; -&gt;*</code>	必须是成员
<code>+= -= /= *= ^= &amp;=  = %= &gt;&gt;= &lt;&lt;=</code>	成员
所有其他二元运算符	非成员

#### 12.5 重载赋值符

赋值符常引起 C++ 程序员初学者的混淆。这是毫无疑问的，因为 ‘=’ 在编程中是最基本的运算符，是在机器层上拷贝寄存器。另外，当使用 ‘=’ 时也能引起拷贝构造函数（第 11 章内容）调用：

① 参见 Rob Murray 所著《C++ Strategies & Tactics》(Addison-Wesley), 1993, 第 47 页。

```
MyType b;
MyType a = b;
a = b;
```

第2行定义了对象**a**。一个新对象先前不存在，现在正被创建。因为现在知道了C++编译器关于对象初始化是如何保护的，所以知道在对象被定义的地方构造函数总是必须被调用。但是是调用哪个构造函数呢？**a**是从现有的**MyType**对象创建的（**b**在等号的右侧），所以只有一个选择：拷贝构造函数。所以虽然这里包括一个等号，但拷贝构造函数仍被调用。

第3行情况就不同了。在等号左侧有一个以前初始化了的对象。很清楚，不用为一个已经存在的对象调用构造函数。在这种情况下，为**a**调用**MyType::operator =**，把出现在右侧的任何东西作为参数（可以有多种取不同右侧参数的**operator=**函数）。

对于拷贝构造函数则没有这个限制。在任何时候使用一个“=”代替普通形式的构造函数调用来初始化一个对象时，无论等号右侧是什么，编译器都会寻找一个接受右边类型的构造函数：

```
//: C12:CopyingVsInitialization.cpp
class Fi {
public:
    Fi() {}
};
class Fee {
public:
    Fee(int) {}
    Fee(const Fi&) {}
};

int main() {
    Fee fee = 1; // Fee(int)
    Fi fi;
    Fee fum = fi; // Fee(Fi)
} ///:~
```

当处理“=”时，记住这个差别是非常重要的：如果对象还没有被创建，初始化是需要的，否则使用赋值**operator=**。

最好避免编写使用“=”的初始化代码，而是用显式的构造函数形式。等号的两种构造形式变为：

```
Fee fee(1);
Fee fum(fi);
```

这个方法可以避免使读者混淆。

### 12.5.1 operator=的行为

在**Integer.h** 和 **Byte.h**中，可以看到**operator=**仅是成员函数，它密切地与“=”左侧的对象相联系。如果允许定义**operator=**为全局的，那么我们会试图重新定义内置的“=”：

```
int operator=(int, MyType); // Global = not allowed!
```

编译器通过强制**operator=**为成员函数而避开这个问题。

当创建一个**operator=**时，必须从右侧对象中拷贝所有需要的信息到当前的对象（即调用

运算符的对象) 以完成为类的“赋值”，对于简单的对象，这是显然的：

```
//: C12:SimpleAssignment.cpp
// Simple operator=()
#include <iostream>
using namespace std;

class Value {
    int a, b;
    float c;
public:
    Value(int aa = 0, int bb = 0, float cc = 0.0)
        : a(aa), b(bb), c(cc) {}
    Value& operator=(const Value& rv) {
        a = rv.a;
        b = rv.b;
        c = rv.c;
        return *this;
    }
    friend ostream&
    operator<<(ostream& os, const Value& rv) {
        return os << "a = " << rv.a << ", b = "
            << rv.b << ", c = " << rv.c;
    }
};

int main() {
    Value a, b(1, 2, 3.3);
    cout << "a: " << a << endl;
    cout << "b: " << b << endl;
    a = b;
    cout << "a after assignment: " << a << endl;
} ///:~
```

这里，“=”左侧的对象拷贝了右侧对象中的所有内容，然后返回它的引用，所以还可以创建更加复杂的表达式。

这个例子犯了一个常见的错误。当准备给两个相同类型的对象赋值时，应该首先检查一下自赋值(self-assignment)：这个对象是否对自身赋值了？在一些情况下，例如本例，无论如何执行这些赋值运算都是无害的，但如果对类的实现进行了修改，那么将会出现差异。如果我们习惯于不做检查，就可能忘记并产生难以发现的错误。

#### 12.5.1.1 类中指针

如果对象不是如此简单时将会发生什么问题？例如，如果对象里包含指向别的对象的指针将如何？简单地拷贝一个指针意味着以指向相同的存储单元的对象而结束。在这种情况下，就需要自己做簿记。

这里有两个解决办法。当做一个赋值运算或一个拷贝构造函数时，最简单的方法是拷贝这个指针所涉及的一切，这是非常直接的。

```
//: C12:CopyingWithPointers.cpp
// Solving the pointer aliasing problem by
// duplicating what is pointed to during
// assignment and copy-construction.
#include "../require.h"
```



```

#include <string>
#include <iostream>
using namespace std;

class Dog {
    string nm;
public:
    Dog(const string& name) : nm(name) {
        cout << "Creating Dog: " << *this << endl;
    }
    // Synthesized copy-constructor & operator=
    // are correct.
    // Create a Dog from a Dog pointer:
    Dog(const Dog* dp, const string& msg)
        : nm(dp->nm + msg) {
        cout << "Copied dog " << *this << " from "
            << *dp << endl;
    }
    ~Dog() {
        cout << "Deleting Dog: " << *this << endl;
    }
    void rename(const string& newName) {
        nm = newName;
        cout << "Dog renamed to: " << *this << endl;
    }
    friend ostream&
    operator<<(ostream& os, const Dog& d) {
        return os << "[" << d.nm << "]";
    }
};

class DogHouse {
    Dog* p;
    string houseName;
public:
    DogHouse(Dog* dog, const string& house)
        : p(dog), houseName(house) {}
    DogHouse(const DogHouse& dh)
        : p(new Dog(dh.p, " copy-constructed")),
          houseName(dh.houseName
              + " copy-constructed") {}
    DogHouse& operator=(const DogHouse& dh) {
        // Check for self-assignment:
        if(&dh != this) {
            p = new Dog(dh.p, " assigned");
            houseName = dh.houseName + " assigned";
        }
        return *this;
    }
    void renameHouse(const string& newName) {
        houseName = newName;
    }
    Dog* getDog() const { return p; }
    ~DogHouse() { delete p; }
    friend ostream&
    operator<<(ostream& os, const DogHouse& dh) {

```



```

        return os << "[" << dh.houseName
            << "]" contains " << *dh.p;
    }
};

int main() {
    DogHouse fidos(new Dog("Fido"), "FidoHouse");
    cout << fidos << endl;
    DogHouse fidos2 = fidos; // Copy construction
    cout << fidos2 << endl;
    fidos2.getDog()->rename("Spot");
    fidos2.renameHouse("SpotHouse");
    cout << fidos2 << endl;
    fidos = fidos2; // Assignment
    cout << fidos << endl;
    fidos.getDog()->rename("Max");
    fidos2.renameHouse("MaxHouse");
} ///:~

```

**Dog**是一个简单的类，仅包含一个用来说明**dog**名字的**string**成员。但是，由于构造函数和析构函数在它们被调用的时候打印信息，所以就可以知道对**Dog**进行操作的时间。注意这第二个构造函数有点像拷贝构造函数，除了它的参数是一个指向**Dog**对象的指针而不是一个引用外，并且它还有第二个参数，是同**Dog**参数的名字相关联的信息。这被用于帮助追踪程序的执行。

可以看到，无论何时成员函数打印信息，它都不是直接获取这些信息的，而是把**\*this** 传送给**cout**。进而调用**ostream operator<<**。用这种方式进行操作是值得的，因为如果想重新格式化**Dog**信息的显示方式（正如通过增加“[”和“]”所做的），仅需要在一处进行操作。

当类中包含指针时，**DogHouse**含有一个**Dog\***并说明了需要定义的4个函数：所有必需的普通构造函数、拷贝构造函数、**operator=**（无论定义它还是不允许它）和析构函数。对**operator=**当然要检查自赋值，虽然这儿并不一定需要，这实际上减少了改变代码而忘记检查自赋值的可能性。

#### 12.5.1.2 引用计数

在上面的例子中，拷贝构造函数和**operator=**对指针所指向的内容作了一个新的拷贝，并由析构函数删除它。但是，如果对象需要大量的内存或过高的初始化，我们也许想避免这种拷贝。解决这个问题的通常方法称为引用计数(*reference counting*)。可以使一块存储单元具有智能，它知道有多少对象指向它。拷贝构造函数或赋值运算意味着把另外的指针指向现在的存储单元并增加引用记数。消除意味着减小引用记数，如果引用记数为0则意味销毁这个对象。

但如果向这个对象（上例中的**Dog**）执行写入操作将会如何呢？因为不止一个对象使用这个**Dog**，所以当修改自己的**Dog**时，也等于也修改了他人的**Dog**。为了解决这个“别名”问题，经常使用另外一个称为写拷贝(*copy-on-write*)的技术。在向这块存储单元写之前，应该确信没有其他使用它。如果引用记数大于1，在写之前必须拷贝这块存储单元，这样就不会影响他人了。这儿提供了一个简单的引用记数和关于写拷贝的例子：

```

//: C12:ReferenceCounting.cpp
// Reference count, copy-on-write
#include "../require.h"
#include <string>
#include <iostream>

```

```

using namespace std;

class Dog {
    string nm;
    int refcount;
    Dog(const string& name)
        : nm(name), refcount(1) {
        cout << "Creating Dog: " << *this << endl;
    }
    // Prevent assignment:
    Dog& operator=(const Dog& rv);
public:
    // Dogs can only be created on the heap:
    static Dog* make(const string& name) {
        return new Dog(name);
    }
    Dog(const Dog& d)
        : nm(d.nm + " copy"), refcount(1) {
        cout << "Dog copy-constructor: "
            << *this << endl;
    }
    ~Dog() {
        cout << "Deleting Dog: " << *this << endl;
    }
    void attach() {
        ++refcount;
        cout << "Attached Dog: " << *this << endl;
    }
    void detach() {
        require(refcount != 0);
        cout << "Detaching Dog: " << *this << endl;
        // Destroy object if no one is using it:
        if(--refcount == 0) delete this;
    }
    // Conditionally copy this Dog.
    // Call before modifying the Dog, assign
    // resulting pointer to your Dog*.
    Dog* unalias() {
        cout << "Unaliasing Dog: " << *this << endl;
        // Don't duplicate if not aliased:
        if(refcount == 1) return this;
        --refcount;
        // Use copy-constructor to duplicate:
        return new Dog(*this);
    }
    void rename(const string& newName) {
        nm = newName;
        cout << "Dog renamed to: " << *this << endl;
    }
    friend ostream&
    operator<<(ostream& os, const Dog& d) {
        return os << "[" << d.nm << "], rc = "
            << d.refcount;
    }
};

```



```

class DogHouse {
    Dog* p;
    string houseName;
public:
    DogHouse(Dog* dog, const string& house)
        : p(dog), houseName(house) {
        cout << "Created DogHouse: " << *this << endl;
    }
    DogHouse(const DogHouse& dh)
        : p(dh.p),
          houseName("copy-constructed " +
                    dh.houseName) {
        p->attach();
        cout << "DogHouse copy-constructor: "
              << *this << endl;
    }
    DogHouse& operator=(const DogHouse& dh) {
        // Check for self-assignment:
        if(&dh != this) {
            houseName = dh.houseName + " assigned";
            // Clean up what you're using first:
            p->detach();
            p = dh.p; // Like copy-constructor
            p->attach();
        }
        cout << "DogHouse operator= : "
              << *this << endl;
        return *this;
    }
    // Decrement refcount, conditionally destroy
    ~DogHouse() {
        cout << "DogHouse destructor: "
              << *this << endl;
        p->detach();
    }
    void renameHouse(const string& newName) {
        houseName = newName;
    }
    void unalias() { p = p->unalias(); }
    // Copy-on-write. Anytime you modify the
    // contents of the pointer you must
    // first unalias it:
    void renameDog(const string& newName) {
        unalias();
        p->rename(newName);
    }
    // ... or when you allow someone else access:
    Dog* getDog() {
        unalias();
        return p;
    }
    friend ostream&
    operator<<(ostream& os, const DogHouse& dh) {
        return os << "[" << dh.houseName
              << "]" contains " << *dh.p;
    }
}

```



```

};

int main() {
    DogHouse
        fidos(Dog::make("Fido"), "FidoHouse"),
        spots(Dog::make("Spot"), "SpotHouse");
    cout << "Entering copy-construction" << endl;
    DogHouse bobs(fidos);
    cout << "After copy-constructing bobs" << endl;
    cout << "fidos:" << fidos << endl;
    cout << "spots:" << spots << endl;
    cout << "bobs:" << bobs << endl;
    cout << "Entering spots = fidos" << endl;
    spots = fidos;
    cout << "After spots = fidos" << endl;
    cout << "spots:" << spots << endl;
    cout << "Entering self-assignment" << endl;
    bobs = bobs;
    cout << "After self-assignment" << endl;
    cout << "bobs:" << bobs << endl;
    // Comment out the following lines:
    cout << "Entering rename(\"Bob\")" << endl;
    bobs.getDog()->rename("Bob");
    cout << "After rename(\"Bob\")" << endl;
} ///:~

```

类**Dog**是**DogHouse**指向的对象。包含了一个引用记数及控制和读引用记数的函数。同时这里存在一个拷贝构造函数，所以可以从现有的对象创建一个新的**Dog**。

函数**attach()**增加一个**Dog**的引用记数用以指示有另一个对象使用它。函数**detach()**减少引用记数。如果引用记数为0，则说明没有对象使用它，所以通过表达式**delete this**，成员函数销毁它自己的对象。

在进行任何修改（例如为一个**Dog**重命名）之前，必须保证所修改的**Dog**没有被别的对象正在使用。这可以通过调用**DogHouse::unalias()**，它又进而调用**Dog::unalias()**来做到这点。如果引用记数为1（意味着没有别的对象指向这块存储单元），后面这个函数将返回存在的**Dog**指针，但如果引用记数大于1（意味着不止一个对象指向这个**Dog**）就要复制这个**Dog**。

拷贝构造函数给源对象**Dog**赋值**Dog**，而不是创建它自己的存储单元。然后因为现在增加了使用这个存储单元的对象，所以通过调用**Dog::attach()**增加引用记数。

**operator=**处理等号左侧已创建的对象，所以它首先必须通过为**Dog**调用**detach()**来整理这个存储单元。如果没有其他对象使用它，这个老的**Dog**将被销毁。然后**operator=**重复拷贝构造函数的行为。注意它首先检查是否给它本身赋予相同的对象。

析构函数调用**detach()**有条件地销毁**Dog**。

为了实现写拷贝，必须控制所有写存储单元的动作。例如成员函数**renameDog()**允许对这个存储单元修改数值。但它首先必须使用**unalias()**防止修改一个已别名化了的存储单元（超过一个对象使用的存储单元）。如果想从**DogHouse**中产生一个指向**Dog**的指针，首先要对指针调用**unalias()**。

在**main()**中测试了几个必须正确实现引用记数的函数：构造函数、拷贝构造函数、**operator=**和析构函数。在**main()**中也通过C调用**renameDog()**测试了写拷贝。

下面是（一部分重新格式化后的）输出结果：

```

Creating Dog: [Fido], rc = 1
Created DogHouse: [FidoHouse]
    contains [Fido], rc = 1
Creating Dog: [Spot], rc = 1
Created DogHouse: [SpotHouse]
    contains [Spot], rc = 1
Entering copy-construction
Attached Dog: [Fido], rc = 2
DogHouse copy-constructor:
    [copy-constructed FidoHouse]
    contains [Fido], rc = 2
After copy-constructing bobs
fidos:[FidoHouse] contains [Fido], rc = 2
spots:[SpotHouse] contains [Spot], rc = 1
bobs:[copy-constructed FidoHouse]
    contains [Fido], rc = 2
Entering spots = fidos
Detaching Dog: [Spot], rc = 1
Deleting Dog: [Spot], rc = 0
Attached Dog: [Fido], rc = 3
DogHouse operator= : [FidoHouse assigned]
    contains [Fido], rc = 3
After spots = fidos
spots:[FidoHouse assigned] contains [Fido], rc = 3
Entering self-assignment
DogHouse operator= : [copy-constructed FidoHouse]
    contains [Fido], rc = 3
After self-assignment
bobs:[copy-constructed FidoHouse]
    contains [Fido], rc = 3
Entering rename("Bob")
After rename("Bob")
DogHouse destructor: [copy-constructed FidoHouse]
    contains [Fido], rc = 3
Detaching Dog: [Fido], rc = 3
DogHouse destructor: [FidoHouse assigned]
    contains [Fido], rc = 2
Detaching Dog: [Fido], rc = 2
DogHouse destructor: [FidoHouse]
    contains [Fido], rc = 1
Detaching Dog: [Fido], rc = 1
Deleting Dog: [Fido], rc = 0

```

通过研究输出结果、跟踪源代码和程序的测试，将加深对这些技术的理解。

#### 12.5.1.3 自动创建operator=

因为将一个对象赋给另一个相同类型的对象是大多数人可能做的事情，所以如果没有创建**type::operator=(type)**，编译器将自动创建一个。这个运算符行为模仿自动创建的拷贝构造函数的行为：如果类包含对象（或是从别的类继承的），对于这些对象，**operator=**被递归调用。这被称为成员赋值(*memberwise assignment*)。见如下例子：

```

//: C12:AutomaticOperatorEquals.cpp
#include <iostream>
using namespace std;

```

```

class Cargo {
public:
    Cargo& operator=(const Cargo&) {
        cout << "inside Cargo::operator=()" << endl;
        return *this;
    }
};

class Truck {
    Cargo b;
};

int main() {
    Truck a, b;
    a = b; // Prints: "inside Cargo::operator=()"
} ///:~

```

为**Truck**自动生成的**operator=**调用**Cargo::operator=**。

一般我们不会想让编译器做这些。对于复杂的类（尤其是它们包含指针时），应该显式地创建一个**operator=**。如果真的不想让人执行赋值运算，可以把**operator=**声明为**private**函数（除非在类内使用它，否则不必定义它）。

## 12.6 自动类型转换

在C和C++中，如果编译器看到一个表达式或函数调用使用了一个不合适的类型，它经常会执行一个自动类型转换，从现在的类型到所要求的类型。在C++中，可以通过定义自动类型转换函数来为用户定义类型达到相同效果。这些函数有两种类型：特殊类型的构造函数和重载的运算符。

### 12.6.1 构造函数转换

如果定义一个构造函数，这个构造函数能把另一类型对象（或引用）作为它的单个参数，那么这个构造函数允许编译器执行自动类型转换。如下例：

```

//: C12:AutomaticTypeConversion.cpp
// Type conversion constructor
class One {
public:
    One() {}
};

class Two {
public:
    Two(const One&) {}
};

void f(Two) {}

int main() {
    One one;
    f(one); // Wants a Two, has a One
} ///:~

```



当编译器看到`f()`以类**One**的对象为参数调用时,编译器检查`f()`的声明并注意到它需要一个类**Two**的对象作为参数。然后,编译器检查是否有从对象**One**到**Two**的方法。它发现了构造函数**Two::Two(One)**,**Two::Two(One)**被悄悄地调用,结果对象**Two**被传递给`f()`。

在这种情况下,自动类型转换避免了定义两个`f()`重载版本的麻烦。然而,代价是调用**Two**的隐藏构造函数,如果关心`f()`的调用效率的话,那就不要使用这种方法。

#### 12.6.1.1 阻止构造函数转换

有时通过构造函数自动转换类型可能出现問題。为了避开这个麻烦,可以通过在前面加关键字**explicit** (只能用于构造函数)来对上例类**Two**的构造函数进行修改:

```
//: C12:ExplicitKeyword.cpp
// Using the "explicit" keyword
class One {
public:
    One() {}
};

class Two {
public:
    explicit Two(const One&) {}
};

void f(Two) {}

int main() {
    One one;
    //! f(one); // No auto conversion allowed
    f(Two(one)); // OK -- user performs conversion
} ///:~
```

通过使类**Two**的构造函数显式化,编译器被告知不能使用那个构造函数执行任何自动转换(那个类中其他非显式化的构造函数仍可以执行自动类型转换)。如果用户想进行转换必须写出代码。上面代码**f(Two(One))**创建一个从类型**One**到**Two**的临时对象,就像编译器在前面版本中所做的那样。

#### 12.6.2 运算符转换

第二种自动类型转换的方法是通过运算符重载。可以创建一个成员函数,这个函数通过在关键字**operator**后跟随想要转换到的类型的方法,将当前类型转换为希望的类型。这种形式的运算符重载是独特的,因为没有指定一个返回类型——返回类型就是正在重载的运算符的名字。下面是一个例子:

```
//: C12:OperatorOverloadingConversion.cpp
class Three {
    int i;
public:
    Three(int ii = 0, int = 0) : i(ii) {}
};

class Four {
    int x;
public:
```



```

    Four(int xx) : x(xx) {}
    operator Three() const { return Three(x); }
};

void g(Three) {}

int main() {
    Four four(1);
    g(four);
    g(1); // Calls Three(1,0)
} ///:~

```

用构造函数技术，目的类执行转换。然而使用运算符技术，是源类执行转换。构造函数技术的价值是在创建一个新类时为现有系统增加了新的转换途径。然而，创建一个单一参数的构造函数总是定义一个自动类型转换（即使它有不只一个参数也是一样，因为其余的参数将被默认处理），这可能并不是我们所想要的（那种情况下可使用**explicit**来避免）。另外，使用构造函数技术没有办法实现从用户定义类型向内置类型转换，这只有运算符重载可能做到。

#### 12.6.2.1 反身性

使用全局重载运算符而不用成员运算符的最便利的原因之一是在全局版本中的自动类型转换可以针对左右任一操作数，而成员版本必须保证左侧操作数已处于正确的形式。如果想两个操作数都被转换，全局版本可以节省很多代码。下面有一个小例子：

```

//: C12:ReflexivityInOverloading.cpp
class Number {
    int i;
public:
    Number(int ii = 0) : i(ii) {}
    const Number
    operator+(const Number& n) const {
        return Number(i + n.i);
    }
    friend const Number
    operator-(const Number&, const Number&);
};

const Number
operator-(const Number& n1,
          const Number& n2) {
    return Number(n1.i - n2.i);
}

int main() {
    Number a(47), b(11);
    a + b; // OK
    a + 1; // 2nd arg converted to Number
    //! 1 + a; // Wrong! 1st arg not of type Number
    a - b; // OK
    a - 1; // 2nd arg converted to Number
    1 - a; // 1st arg converted to Number
} ///:~

```

类**Number**有一个成员**operator+**和一个**friend operator-**。因为有一个使用单一**int**参数

的构造函数，在正确的条件下，`int`可以自动转换为`Number`。在`main()`里，可以看到增加一个`Number`到另一个`Number`进行得很好，这是因为它重载的运算符非常相匹配。当编译器看到一个`Number`后跟一个`+`号和一个`int`时，它也能和成员函数`Number::operator+`相匹配并且构造函数把`int`参数转换为`Number`。但当编译器看到一个`int`、一个`+`号和一个`Number`时，它就不知道如何去做，因为它所拥有的是`Number::operator+`，需要左侧的操作数是`Number`对象。因此，编译器发出一个出错信息。

对于`friend operator-`，情况就不同了。编译器需要填满两个参数，它不是限定`Number`作为左侧参数。因此，如果看到表达式

```
1 - a
```

编译器就使用构造函数把第一个参数转换为`Number`。

有时也许想通过把它们设成成员函数来限定运算符的使用。例如当用一个矢量与矩阵相乘，矢量必须在右侧。但如果想让运算符转换任一个参数，就要使运算符为友元函数。

幸运的是，编译器不会把表达式`1-1`的两个参数转换为`Number`对象，然后调用`operator-`。那将意味着现有的C代码可能突然执行不同的工作了。编译器首先匹配“最简单的”可能性，对于表达式`1-1`将优先使用内置运算符。

### 12.6.3 类型转换例子

本例中的自动类型转换对于任一含有字符串的类（本例中，因为是简单的，所以使用的是标准C++ `string`类）是非常有帮助的。如果不用自动类型转换就想从标准的C库函数中使用所有的字符串函数，那么就非得为每一个函数写一个相应的成员函数，就像下面的例子：

```
//: C12:Strings1.cpp
// No auto type conversion
#include "../require.h"
#include <cstring>
#include <cstdlib>
#include <string>
using namespace std;

class Stringc {
    string s;
public:
    Stringc(const string& str = "") : s(str) {}
    int strcmp(const Stringc& S) const {
        return ::strcmp(s.c_str(), S.s.c_str());
    }
    // ... etc., for every function in string.h
};

int main() {
    Stringc s1("hello"), s2("there");
    s1 strcmp(s2);
} ///:~
```

这里只写了一个`strcmp()`函数，但必须为可能需要的`<cstring>`中的每一个写一个相应的函数。幸运的是，可以提供一个允许访问`<cstring>`中所有函数的自动类型转换：

```
//: C12:Strings2.cpp
```

```

// With auto type conversion
#include "../require.h"
#include <cstring>
#include <cstdlib>
#include <string>
using namespace std;

class Stringc {
    string s;
public:
    Stringc(const string& str = "") : s(str) {}
    operator const char*() const {
        return s.c_str();
    }
};

int main() {
    Stringc s1("hello"), s2("there");
    strcmp(s1, s2); // Standard C function
    strstr(s1, s2); // Any string function!
} ///:~

```

因为编译器知道如何从**String c**转换到**char\***，所以现在任何一个接受**char\***参数的函数也可以接受**Stringc**参数。

#### 12.6.4 自动类型转换的缺陷

因为编译器必须选择如何执行类型转换，所以如果没有正确地设计出转换，编译器会产生麻烦。类**X**可以用**operator Y()**将它本身转换到类**Y**，这是一个简单且明显的情况。如果类**Y**有一个单个参数为**X**的构造函数，这也表示同样的类型转换。现在编译器有两个从**X**到**Y**的转换方法，所以当发生转换时，编译器会产生一个二义性转换错误：

```

//: C12:TypeConversionAmbiguity.cpp
class Orange; // Class declaration

class Apple {
public:
    operator Orange() const; // Convert Apple to Orange
};

class Orange {
public:
    Orange(Apple); // Convert Apple to Orange
};

void f(Orange) {}

int main() {
    Apple a;
    //! f(a); // Error: ambiguous conversion
} ///:~

```

这个问题的解决方法是不要那样做，而是仅提供单一的从一个类型到另一个类型的自动转换方法。

当提供了转换到不止一种类型的自动转换时，会发生一个更难解决的问题。有时，这个问题被称为扇出(*fan-out*)：

```
//: C12:TypeConversionFanout.cpp
class Orange {};
class Pear {};

class Apple {
public:
    operator Orange() const;
    operator Pear() const;
};

// Overloaded eat():
void eat(Orange);
void eat(Pear);

int main() {
    Apple c;
    //! eat(c);
    // Error: Apple -> Orange or Apple -> Pear ???
} ///:~
```

类**Apple**有向**Orange**和**Pear**的自动转换。这样存在一个隐藏的缺陷：使用了创建的两种版本的重载运算符**eat()**时就出现问题了（只有一个版本时，**main()**里的代码会正常运行）。

通常，对于自动类型的解决方法是只提供一个从某类型向另一个类型转换的自动转换版本。当然也可以有多个向其他类型的转换，但它们不应该是自动转换，而应该用如**makeA()**和**makeB()**这样的名字来创建显式的函数调用。

#### 12.6.4.1 隐藏的行为

自动类型转换会引入比所希望的更多的潜在行为。下面要费点力去理解了，看看**CopyingVsInitialization.cpp**修改后的例子：

```
//: C12:CopyingVsInitialization2.cpp
class Fi {};

class Fee {
public:
    Fee(int) {}
    Fee(const Fi&) {}
};

class Fo {
    int i;
public:
    Fo(int x = 0) : i(x) {}
    operator Fee() const { return Fee(i); }
};

int main() {
    Fo fo;
    Fee fee = fo;
} ///:~
```

这里没有从**Fo**对象创建**Fee fee**的构造函数。然而，**Fo**有一个到**Fee**的自动类型转换。这

里也没有从**Fee**对象创建**Fee**的拷贝构造函数，但这是一种能由编译器帮助我们创建的特殊函数之一（默认的构造函数、拷贝构造函数、**operator=**和析构函数可自动创建）。对于下面正确的声明：

```
Fee fee = fo;
```

自动类型转换运算符被调用并创建一个拷贝函数：

自动类型转换应小心使用。同所有重载的运算符相比，它在减少代码方面是非常出色的，但不值得无缘无故地使用。

## 12.7 小结

运算符重载存在的原因是为了使编程容易。运算符重载没有什么神秘的，它只不过是拥有有趣名字的函数。当它以正确的形式出现时，编译器调用这个函数。但如果运算符重载对于类的设计者或类的使用者不能提供特别显著的益处，则最好不要使用，因为增加运算符重载会使问题混淆。

## 12.8 练习

部分练习题的答案可以在本书的电子文档“*Annotated Solution Guide for Thinking in C++*”中找到，只需支付很少的费用就可以从<http://www.BruceEckel.com>得到这个电子文档。

- 12-1 写一个有重载**operator++**的类。试着用前缀和后缀两种形式调用此运算符，看看编译器会给出什么警告。
- 12-2 创建含有一个**int**成员的简单的类，以成员函数的形式重载**operator+**。同时提供一个**print()**成员函数，以**ostream&**作为参数并打印出该**ostream&**。测试该类表明它可以正确运行。
- 12-3 用成员函数形式，在练习2的类中增加一个二元**operator-**。要求可以在**a+b-c**这样复杂的表达式中使用该类的对象。
- 12-4 在练习2的例子中增加**operator++**和**operator--**，要包括前缀和后缀版本，使得它们返回自增和自减对象。确保后缀版本返回正确的值。
- 12-5 修改练习4的自增和自减运算符，以使得前缀版本使用非常量而后缀版本使用常量。显示它们运行正确并解释为什么实际中要这么做。
- 12-6 改变练习2中的**print()**函数，使得它是重载**operator<<**，就像在**IostreamOperatorOverloading.cpp**中一样。
- 12-7 修改练习3，使得**operator+**和**operator-**是非成员函数。表明它们仍能正确运行。
- 12-8 对练习2的类中增加一元**operator-**，并表明它可以正确运行。
- 12-9 写一个只含有单个**private char**成员的类。重载**iostream operator<<**和**>>**（像在**IostreamOperatorOverloading.cpp**中的一样）并测试它们，可以用**fstreams**、**stringstreams**和**cin**与**cout**测试它们。
- 12-10 测定为了前缀**operator++**和**operator--**，编译器传递的哑常量值。
- 12-11 写一个包含一个**double**成员的**Number**类，并增添重载的**operator+**、**-**、**\***、**/**和赋值符。为这些函数合理地选择返回值以便可以链式写表达式，以提高效率。写一个自动类型转换**operator double()**。

- 12-12 如果返回值优化还没有被使用, 修改练习11, 使用返回值优化。
- 12-13 创建一个包含指针的类, 表明如果允许编译器生成`operator=`, 结果将得到具有不同别名、指向同一存储区域的指针。现在通过定义自己的`operator=`解决这个问题, 并表明它纠正了别名。确保检查自赋值并完全处理了此问题。
- 12-14 写一个包含`string`和`static int`成员的类`Bird`。在预设的构造函数中, 根据类的名字(`Bird #1`, `Bird #2`, 等等), 使用`int`成员自动地生成一个置于`string`中的标识符。为`ostream`增加`operator<<`以打印`Bird`对象。写一个赋值`operator=`和一个拷贝构造函数。在`main()`中, 验证它们都可正确运行。
- 12-15 写一个类`BirdHouse`, 包含一个对象、一个指针和一个练习14中类`Bird`的引用。构造函数的参数是三个`Bird`类对象。为`BirdHouse`增加`ostream operator<<`。写一个赋值`operator=`和一个拷贝构造函数。在`main()`中, 验证它们都正确地运行。确保能对`BirdHouse`对象链接赋值运算, 并生成包括多种操作的表达式。
- 12-16 对练习15中的类`Bird`和`BirdHouse`增加一个`int`数据成员。增加成员运算符`+`、`-`、`*`和`/`, 它们使用`int`成员在各自的成员上进行操作。验证这些工作。
- 12-17 用非成员运算符进行练习16的操作。
- 12-18 增加`operator--`到`SmartPointer.cpp`和`NestedSmartPointer.cpp`中。
- 12-19 修改`CopyingVsInitialization.cpp`, 使得所有的构造函数打印出正在进行操作的信息。现在验证拷贝构造函数的两种调用形式是相等的。
- 12-20 试着为一个类创建一个非成员`operator=`, 并看看能得到编译器的何种信息。
- 12-21 用拷贝构造函数创建一个类, 该拷贝构造函数的第二个参数是一个预设值为“op=call.”的`string`成员。创建一个函数, 它通过传值方式接受此类的对象, 并表明拷贝构造函数被正确地调用了。
- 12-22 在`CopyingWithPointers.cpp`中, 删除`DogHouse`中的`operator=`, 表明编译器生成的`operator=`正确地拷贝了`string`成员, 但简单地给`Dog`指针起了别名。
- 12-23 在`ReferenceCounting.cpp`中, 增加一个`static int`成员和一个基本`int`成员作为`Dog`和`DogHouse`的数据成员。在两个类的所有构造函数中, 把`static int`成员加1并把结果赋予基本`int`成员以记录所创建的对象数目。进行必要的修改, 打印出涉及的对象的`int`标识符。
- 12-24 创建包含一个`string`数据成员的类。在构造函数中初始化`string`成员, 但不用创建拷贝构造函数或`operator=`。创建第二个类, 其成员对象是第一个类的对象; 同样不要为这个类创建拷贝构造函数或`operator=`。表明拷贝构造函数和`operator=`可被编译器正确地生成。
- 12-25 合并到`OverloadingUnaryOperators.cpp`和`Integer.cpp`中的类。
- 12-26 通过新增加两个`Dog`中的成员函数, 它们无参数并返回为`void`值, 来修改`PointerToMemberOperator.cpp`。创建并测试作用于这两个新函数上的重载`operator->*`。
- 12-27 在`NestedSmartPointer.cpp`中增加`operator->*`。
- 12-28 创建两个类`Apple`和`Orange`。在类`Apple`中, 创建一个构造函数, 其参数是`Orange`类的对象。然后创建一个函数, 它的参数是`Apple`类对象, 并用`Orange`类对象调用该函数。现在显式应用`Apple`类的构造函数以表明自动类型转换被禁止。修改对函数的调

用，使得能成功地做显式转换。

- 12-29 在**ReflexivityInOverloading.cpp**中增加一个全局**operator\***并表明它具有反身性。
- 12-30 创建两个类并创建**operator+**和转换函数，使得对于这两个类，加法具有反身性。
- 12-31 不用自动转换运算符，而通过创建一个显式的执行类型转换的函数修改**TypeConversion-Fanout.cpp**。
- 12-32 写一段简单的代码，在其中对**double**类型使用**operator+**、**-**、**\***和**/**。看看编译器如何生成汇编代码并根据生成的汇编语言找出并解释发生了什么。



## 动态对象创建

有时我们能知道程序中对象的确切数量、类型和生命期。但情况并不总是这样。

空中交通指挥系统将会需要处理多少架飞机？一个CAD系统将会需要多少个形体？在一个网络中将会有多少个节点？

为了解决这个普通的编程问题，在运行时可以创建和销毁对象是最基本的要求。当然，C早就提供了动态内存分配（*dynamic memory allocation*）函数`malloc()`和`free()`（以及`malloc()`的变体），这些函数在运行时从堆（也称自由内存）中分配存储单元。

然而，在C++中这些函数将不能很好地运行。因为构造函数不允许我们向它传递内存地址来进行初始化。如果那么做了，我们可能：

- 1) 忘记了。则在C++中有保证的对象初始化将会难以保证。
- 2) 期望发生正确的事，但在对对象进行初始化之前意外地对对象进行了某种操作。
- 3) 把错误规模的对象传递给它。

当然，即使我们正确地完成了每件事，修改我们程序的人也容易犯同样的错误。不正确的初始化要对大部分编程问题承担责任，所以在堆上创建对象时确保构造函数调用是特别重要的。

C++是如何保证正确的初始化和清理，又允许我们在堆上动态创建对象呢？

答案是，使动态对象创建成为语言的核心。`malloc()`和`free()`是库函数，因此不在编译器控制范围之内。然而，如果我们有一个完成动态内存分配及初始化组合动作的运算符和另一个完成清理及释放内存组合动作的运算符，编译器仍可以保证所有对象的构造函数和析构函数都会被调用。

在本章中，我们将明白C++的`new`和`delete`是如何通过在堆上安全地创建对象来出色地解决这个问题。

### 13.1 对象创建

当创建一个C++对象时，会发生两件事：

- 1) 为对象分配内存。
- 2) 调用构造函数来初始化那个内存。

到目前为止，应该确保步骤2一定发生。C++强迫这样做是因为未初始化的对象是程序出错的主要原因。对象在哪里和如何被创建无关紧要——构造函数总是需要被调用。

然而，步骤1可以用几种方式或在可选择的时间发生：

- 1) 在静态存储区域，存储空间在程序开始之前就可以分配。这个存储空间在程序的整个运行期间都存在。
- 2) 无论何时到达一个特殊的执行点（左大括号）时，存储单元都可以在栈上被创建。出了执行点（右大括号），这个存储单元自动被释放。这些栈分配运算内置于处理器的指



令集中，非常有效。然而，在写程序的时候，必须知道需要多少个存储单元，以便编译器生成正确的指令。

- 3) 存储单元也可以从一块称为堆（也被称为自由存储单元）的地方分配。这被称为动态内存分配。在运行时调用程序分配这些内存。这意味着可以在任何时候决定分配内存及分配多少内存。当然也需负责决定何时释放内存。这块内存的生存期由我们选择决定——而不受范围决定。

这三个区域经常被放在一块连续的物理存储单元里：静态内存、栈和堆（由编译器的开发者决定它们的顺序），但没有一定的规则。堆栈可以在特定的地方，堆的实现可以通过调用由运算系统分配的一块存储单元。这三件事无需程序设计者来完成。当申请内存的时候，只要知道它们在哪里就行了。

### 13.1.1 C从堆中获取存储单元的方法

为了在运行时动态分配内存，C在它的标准库函数中提供了一些函数：从堆中申请内存的函数**malloc()**以及它的变种**calloc()**和**realloc()**、释放内存返回给堆的函数**free()**。这些函数是有效的但较原始的，需要编程人员理解和小心使用。为了使用C的动态内存分配函数在堆上创建一个类的实例，我们必须这样做：

```
//: C13:MallocClass.cpp
// Malloc with class objects
// What you'd have to do if not for "new"
#include "../require.h"
#include <cstdlib> // malloc() & free()
#include <cstring> // memset()
#include <iostream>
using namespace std;

class Obj {
    int i, j, k;
    enum { sz = 100 };
    char buf[sz];
public:
    void initialize() { // Can't use constructor
        cout << "initializing Obj" << endl;
        i = j = k = 0;
        memset(buf, 0, sz);
    }
    void destroy() const { // Can't use destructor
        cout << "destroying Obj" << endl;
    }
};

int main() {
    Obj* obj = (Obj*)malloc(sizeof(Obj));
    require(obj != 0);
    obj->initialize();
    // ... sometime later:
    obj->destroy();
    free(obj);
} ///:~
```

在下面这行代码中，使用了**malloc()**为对象分配内存：



```
Obj* obj = (Obj*)malloc(sizeof(Obj));
```

这里用户必须决定对象的长度（这也是程序出错原因之一）。由于`malloc()`只是分配了一块内存而不是生成一个对象，所以它返回了一个`void*`类型指针。而C++不允许将一个`void*`类型指针赋予任何其他指针，所以必须做类型转换。

因为`malloc()`可能找不到可分配的内存（在这种情况下它返回0），所以必须检查返回的指针以确保内存分配成功。

但这一行最易出现问题：

```
Obj->initialize();
```

用户在使用对象之前必须记住对它初始化。注意构造函数没有被使用，这是因为构造函数不能被显式地调用<sup>①</sup>——它是在对象创建时由编译器调用。问题是现在用户可能在使用对象时忘记执行初始化，因此这又是一个程序出错的主要来源。

许多程序设计者发现C的动态内存分配函数太复杂，容易令人混淆。所以，C程序设计者常常在静态内存区域使用虚拟内存机制分配很大的变量数组以避免使用动态内存分配。为了在C++中使得一般的程序员可以安全使用库函数而不费力，所以C的动态内存方法是不可接受的。

### 13.1.2 operator new

C++中的解决方案是把创建一个对象所需的所有动作都结合在一个称为`new`的运算符里。当用`new`（`new`的表达式）创建一个对象时，它就在堆里为对象分配内存并为这块内存调用构造函数。因此，如果写出下面的表达式：

```
MyType *fp = new MyType(1,2);
```

在运行时等价于调用`malloc(sizeof(MyType))`（常常，就是精确地调用`malloc()`），并使用`(1, 2)`作为参数表来为`MyType`调用构造函数，`this`指针指向返回值的地址。在指针被赋给`fp`之前，它是不定的、初始化的对象——在这之前我们甚至不能触及它。它自动地被赋予正确的`MyType`类型，所以不必进行映射。

默认的`new`还进行检查以确信在传递地址给构造函数之前内存分配是成功的，所以不必显式地确定调用是否成功。在本章后面，我们将会发现，如果没有可供分配的内存会发生什么事情。

我们可以为类使用任何可用的构造函数而写一个`new`表达式。如果构造函数没有参数，可以写没有构造函数参数表的`new`表达式：

```
MyType *fp = new MyType;
```

我们已经注意到了，在堆里创建对象的过程变得简单了——只是一个简单的表达式，它带有内置的长度计算、类型转换和安全检查。这样在堆里创建一个对象和在栈里创建一个对象一样容易。

### 13.1.3 operator delete

`new`表达式的反面是`delete`表达式。`delete`表达式首先调用析构函数，然后释放内存（经常是调用`free()`）。正如`new`表达式返回一个指向对象的指针一样，`delete`表达式需要一个对象的地址。

① 这里，称为定位`new`（*placement new*）的特殊语法可用来在一块预先分配好的内存上调用构造函数。这将在后面的章节中加以介绍。

```
delete fp;
```

上面的表达式清除了早先创建的动态分配的**MyType**类型对象。

**delete**只用于删除由**new**创建的对象。如果用**malloc()** (或**calloc()**或**realloc()**) 创建一个对象, 然后用**delete**删除它, 这个动作行为是未定义的。因为大多数默认的**new**和**delete**实现机制都使用了**malloc()**和**free()**, 所以很可能会没有调用析构函数就释放了内存。

如果正在删除的对象的指针是0, 将不发生任何事情。为此, 人们经常建议在删除指针后立即把指针赋值为0以免对它删除两次。对一个对象删除两次可能会产生某些问题。

#### 13.1.4 一个简单的例子

这个例子显示了初始化发生的情况:

```
//: C13:Tree.h
#ifndef TREE_H
#define TREE_H
#include <iostream>

class Tree {
    int height;
public:
    Tree(int treeHeight) : height(treeHeight) {}
    ~Tree() { std::cout << " "; }
    friend std::ostream&
    operator<<(std::ostream& os, const Tree* t) {
        return os << "Tree height is: "
            << t->height << std::endl;
    }
};
#endif // TREE_H ///:~

//: C13:NewAndDelete.cpp
// Simple demo of new & delete
#include "Tree.h"
using namespace std;

int main() {
    Tree* t = new Tree(40);
    cout << t;
    delete t;
} ///:~
```

我们通过打印**Tree**的值得知构造函数被调用了。这里是通过调用参数为**ostream**和**Tree\***类型的重载**operator<<**来实现这个运算的。注意, 虽然这个函数被声明为一个友元 (**friend**), 但它还是被定义为一个内联函数。这仅仅是出于方便考虑——定义一个友元函数为内联函数不会改变友元状态, 而且它仍是全局函数而不是一个类的成员函数。也要注意返回值是整个输出表达式的结果, 它本身是一个**ostream&** (为了满足函数返回值类型, 它必须是**ostream&**)。

#### 13.1.5 内存管理的开销

当在堆栈里自动创建对象时, 对象的大小和它们的生存期被准确地内置在生成的代码里,

这是因为编译器知道确切的类型、数量和范围。而在堆里创建对象还包括另外的时间和空间的开销。以下是一个典型的情况。(可以用`calloc()`或`realloc()`代替`malloc()`)

调用`malloc()`，这个函数从堆里申请一块内存（这实际上是用`malloc()`代码的一部分）。

从堆里搜索一块足够大的内存来满足请求。这可以通过检查按某种方式排列的映射或目录来实现，这样的映射或目录用以显示内存的使用情况。这个过程很快但可能要试探几次，所以它可能是不确定的——即每次运行`malloc()`并不是花费了完全相同的时间。

在指向这块内存的指针返回之前，这块内存的大小和地址必须记录下来，这样以后调用`malloc()`就不会使用它，而且当调用`free()`时，系统就会知道释放多大的内存。

实现这些运算的方法可能变化很大。例如，不能阻止处理器中的内存分配原语的执行。如果好奇的话，可以写一个测试程序来估计`malloc()`实现的方法。如果有的话，当然也可以读库函数的源代码。(GNU C 的源代码总是有的。)

## 13.2 重新设计前面的例子

使用`new`和`delete`，对于本书前面介绍的`Stash`例子，可以使用到目前为止讨论的所有的技术来重写。检查这个新代码将有助于对这些主题的复习。

在本书的此处，类`Stash`和`Stack`自己都将不“拥有”它们指向的对象。即当`Stash`或`Stack`对象出了范围，它也不会为它指向的对象调用`delete`。试图使它们成为普通的类是不可能的，原因是它们是`void`指针。如果`delete`一个`void`指针，惟一发生的事就是释放了内存，这是因为既没有类型信息也没有办法使得编译器知道要调用哪个析构函数。

### 13.2.1 使用`delete void*`可能会出错

如果想对一个`void*`类型指针进行`delete`操作，要注意这将可能成为一个程序错误，除非指针所指的内容是非常简单的，因为，它将不执行析构函数。下面的例子将显示发生的情况：

```
//: C13:BadVoidPointerDeletion.cpp
// Deleting void pointers can cause memory leaks
#include <iostream>
using namespace std;

class Object {
    void* data; // Some storage
    const int size;
    const char id;
public:
    Object(int sz, char c) : size(sz), id(c) {
        data = new char[size];
        cout << "Constructing object " << id
              << ", size = " << size << endl;
    }
    ~Object() {
        cout << "Destructing object " << id << endl;
        delete []data; // OK, just releases storage,
        // no destructor calls are necessary
    }
};
```



```
int main() {
    Object* a = new Object(40, 'a');
    delete a;
    void* b = new Object(40, 'b');
    delete b;
} ///:~
```

类**Object**包含了一个**void\***指针，它被初始化指向“元”数据(它没有指向含有析构函数的对象)。在**Object**的析构函数中，对这个**void\***指针调用**delete**并不会发生什么错误，因为所需要的仅是释放这块内存。

但在**main()**中，我们看到使**delete**知道它所操作的对象类型是十分有必要的。输出如下：

```
Constructing object a, size = 40
Destructing object a
Constructing object b, size = 40
```

因为**delete a**知道**a**指向一个**Object**对象，所以析构函数将会被调用，从而释放了分配给**data**的内存。但是，正如在进行**delete b**的操作中，如果通过**void\***类型的指针对一个对象进行操作，则只会释放**Object**对象的内存，而不会调用析构函数，也就不会释放**data**所指向的内存。编译这个程序时，编译器会认为我们知道所做的一切。于是我们不会看到任何警告信息。但因此我们会丢失大量的可用内存。

如果在程序中发现内存丢失的情况，那么就搜索所有的**delete**语句并检查被删除指针的类型。如果是**void\***类型，则可能发现了引起内存丢失的某个因素（因为C++还有很多其他的引起内存丢失的因素）。

### 13.2.2 对指针的清除责任

为了使**Stash**和**Stack**容器具有灵活性（可以包含任意类型的对象），要使用**void**指针。这意味着当一个指针从**Stash**或**Stack**对象返回时，必须在使用之前把它转换为适当的类型。如上所示，在删除它之前也必须把它转换为适当的类型，否则将会丢失内存。

解决内存泄漏的另一个工作在于确保对容器中的每一个对象调用**delete**。容器含有**void\***类型指针，因此不能正确地执行清除，所以容器自己不能“管理”指针。于是用户必须负责清除这些对象。如果把指向在栈上创建的对象指针和指向在堆上创建的对象指针都存放在同一个容器中，将会发生严重的问题。（当从容器中取回一个指针时，我们如何才能知道它所指向的对象是被分配在哪块内存上的呢？）因此不管是通过精心的设计或是通过只作用在堆上的类创建，我们都必须保证存储在如下版本的**Stash**和**Stack**上的对象仅是在堆上创建的。

保证由客户程序员负责清除容器中的所有指针同样是很重要的。在前面的例子中，已经看到**Stack**类是如何在它的析构函数中检查所有的**Link**对象已经出栈了的。但对于**Stash**，需要使用另一种方法。

### 13.2.3 指针的Stash

**Stash**的新版本称为**PStash**，它含有在堆中本来就存在的对象的指针。而前面章节中旧的**Stash**则是通过传值方式拷贝对象到**Stash**的容器。使用**new**和**delete**，控制指向在堆中创建的

对象的指针就变得安全、容易了。

下面提供了“**pointer Stash**”的头文件：

```
//: C13:PStash.h
// Holds pointers instead of objects
#ifndef PSTASH_H
#define PSTASH_H

class PStash {
    int quantity; // Number of storage spaces
    int next; // Next empty space
    // Pointer storage:
    void** storage;
    void inflate(int increase);
public:
    PStash() : quantity(0), storage(0), next(0) {}
    ~PStash();
    int add(void* element);
    void* operator[](int index) const; // Fetch
    // Remove the reference from this PStash:
    void* remove(int index);
    // Number of elements in Stash:
    int count() const { return next; }
};
#endif // PSTASH_H ///:~
```

基本的数据成分是非常相似的，但现在**storage**是一个**void**指针数组，并且用**new**代替**malloc()**为这个数组分配内存。在下面这个表达式中：

```
void** st = new void*[quantity + increase];
```

对象的类型是**void\***，所以这个表达式表示分配了一个**void**指针的数组。

析构函数删除**void**指针本身，而不是试图删除它们所指向的内容（正如前面所指出的，释放它们的内存但不调用析构函数，这是因为一个**void**指针没有类型信息）。

其他方面的变化是用**operator[]**代替了函数**fetch()**，这在语句构成上显得更有意义。因为返回一个**void\***指针，所以用户必须记住在容器内存储的是什么类型，在取回它们时要对这些指针进行类型转换（这是在以后章节中将要修改的问题）。

下面是成员函数的定义：

```
//: C13:PStash.cpp {0}
// Pointer Stash definitions
#include "PStash.h"
#include "../require.h"
#include <iostream>
#include <cstring> // 'mem' functions
using namespace std;

int PStash::add(void* element) {
    const int inflateSize = 10;
    if(next >= quantity)
        inflate(inflateSize);
    storage[next++] = element;
    return(next - 1); // Index number
}
```



```

// No ownership:
PStash::~PStash() {
    for(int i = 0; i < next; i++)
        require(storage[i] == 0,
            "PStash not cleaned up");
    delete []storage;
}

// Operator overloading replacement for fetch
void* PStash::operator[](int index) const {
    require(index >= 0,
        "PStash::operator[] index negative");
    if(index >= next)
        return 0; // To indicate the end
    // Produce pointer to desired element:
    return storage[index];
}

void* PStash::remove(int index) {
    void* v = operator[](index);
    // "Remove" the pointer:
    if(v != 0) storage[index] = 0;
    return v;
}

void PStash::inflate(int increase) {
    const int psz = sizeof(void*);
    void** st = new void*[quantity + increase];
    memset(st, 0, (quantity + increase) * psz);
    memcpy(st, storage, quantity * psz);
    quantity += increase;
    delete []storage; // Old storage
    storage = st; // Point to new memory
} ///:~

```

除了用储存指针代替整个对象的拷贝外，函数**add()**的效果和以前是一样的。

**inflate()**的代码被修改为能处理**void\***指针数组的存储，而不是之前的设计，只处理元比特。这里没有优先使用数组索引的拷贝方法，而是使用标准C库函数中的**memset()**来使所有新的内存置0(并不是一定要如此,因为**PStash**有可能正确地管理所有的内存，但小心点是没有害处的)，然后用**memcpy()**把存在的数据从原来的地方移到一个新的地方。通常类似于**memset()**和**memcpy()**的函数随着时间会逐渐优化，所以它们会比前面所示的循环更快。但由于类似**inflate()**的函数可能没有被使用，所以一般看不出性能上的差异。然而这种比循环更简练的函数调用有助于防止编码错误。

为了由客户程序完全负责对象的清除，有两种方法可以获得**PStash**中的指针：其一是使用**operator[]**，它简单地返回作为一个容器成员的指针。第二种方法是使用成员函数**remove()**，它返回指针，并且通过置0的方法从容器中删除该指针。当**PStash**的析构函数被调用时，它进行检查以确信所有的对象指针已被删除。如果注意到指针还没有被删除，则可以通过删除它来防止内存丢失（后面的章节中有更加智能的方法）。

#### 13.2.3.1 一个测试程序

为了测试**PStash**，我们重写了**Stash**的测试程序：

```

//: C13:PStashTest.cpp
//{L} PStash
// Test of pointer Stash
#include "PStash.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
    PStash intStash;
    // 'new' works with built-in types, too. Note
    // the "pseudo-constructor" syntax:
    for(int i = 0; i < 25; i++)
        intStash.add(new int(i));
    for(int j = 0; j < intStash.count(); j++)
        cout << "intStash[" << j << "] = "
             << *(int*)intStash[j] << endl;
    // Clean up:
    for(int k = 0; k < intStash.count(); k++)
        delete intStash.remove(k);
    ifstream in ("PStashTest.cpp");
    assure(in, "PStashTest.cpp");
    PStash stringStash;
    string line;
    while(getline(in, line))
        stringStash.add(new string(line));
    // Print out the strings:
    for(int u = 0; stringStash[u]; u++)
        cout << "stringStash[" << u << "] = "
             << *(string*)stringStash[u] << endl;
    // Clean up:
    for(int v = 0; v < stringStash.count(); v++)
        delete (string*)stringStash.remove(v);
} ///:~

```

与前面一样，我们创建了**Stash**对象，并且为它们加入了内容。不同的是这次的内容是由**new**表达式产生的指针。首先请注意这一行：

```
intStash.add(new int(i));
```

这个表达式**new int(i)**使用了伪构造函数形式，因此将在堆上创建了一块区域用来存储这个新的**int**对象，同时这个**int**对象被初始化为**i**。

打印时，由**PStash::operator[]**返回的值必须被转换为正确的类型，对于这个程序其余的**PStash**对象，也将重复这个动作。这是使用**void**指针的缺点，将在后面的章节中解决。

测试的第2步是打开源程序文件，并逐行把它读到每一个**PStash**里。首先用**getline()**把每一行读入一个**String**对象，然后对**line**进行**new string**操作，将这一行的内容拷贝下来。如果每次只是传送**line**的地址，将会得到指向**line**的一些指针，而此时**line**仅包含了所读文件的最后一行的内容。

当取回指针时，我们可以看到表达式：

```
*(string*)stringStash[v]
```



为了使`operator[]`返回的指针具有正确的类型，它们必须被转换为`String*`。然后，`String*`被间接引用，所以此表达式的计算结果相当于一个对象，这时编译器也认为一个`string`对象被发送给了`cout`。

在堆上创建的对象必须通过`remove()`语句进行注销，否则将会实时地得到一个信息，告诉我们并没有完全清除那些在`PStash`中的对象。注意，对于`int`指针，类型转换不是必需的，因为`int`类的对象没有析构函数，我们所需要的仅是释放内存。

```
delete intStash.remove(k);
```

但是，对于`string`指针，如果忘记了类型转换，则会出现内存泄漏的情况。所以说进行类型转换是十分重要的。

```
delete (string*)stringStash.remove(k);
```

这些问题的一部分（但不是全部）可以使用模板进行解决。（我们将在第16章中学习模板）。

### 13.3 用于数组的`new`和`delete`

在栈或堆上创建一个对象数组是同样容易的。当然，应当为数组里的每一个对象调用构造函数。但这里有一个限制条件：由于不带参数的构造函数必须被每一个对象调用，所以除了在栈上聚合初始化（见第6章）外还必须有一个默认的构造函数。

当使用`new`在堆上创建对象数组时，还必须多做一些操作。下面是一个创建对象数组的例子：

```
MyType* fp = new MyType[100];
```

这样在堆上为100个`MyType`对象分配了足够的内存并为每一个对象调用了构造函数。但是现在，仅拥有一个`MyType*`。它和用下面的表达式创建单个对象得到的结果是一样的：

```
MyType* fp2 = new MyType;
```

因为这是我们写的代码，所以我们知道`fp`实际上是一个数组的起始地址，所以可以使用类似于`fp[3]`的形式来选择数组的元素。但销毁这个数组时发生了什么呢？下面的语句看起来是完全一样的：

```
delete fp2; // OK
delete fp;  // Not the desired effect
```

并且它们的效果也应该是一样：为所给地址指向的`MyType`对象调用析构函数，然后释放内存。对于`fp2`，这样是正确的，但对于`fp`，另外99个析构函数没有调用。适当数量的存储单元会被释放，但是，由于它们被分配在一个整块的内存中，而整个内存块的大小被分配程序存储在某个地方。

解决办法是给编译器一个信息，说明它实际上是一个数组的起始地址。这可以用下面的语法来实现：

```
delete []fp;
```

空的方括号告诉编译器产生代码，该代码的任务是将从数组创建时存放在某处的对象数量取回，并为数组的所有对象调用析构函数。这实际上是对以前形式的改良，我们偶尔仍可

以在旧版本中看到如下的代码：

```
delete [100]fp;
```

这个语法强迫程序设计者加入数组中对象的数量，但程序设计者有可能把对象的数量弄错。而让编译器处理这件事引起的附加代价是很低的，所以只在一个地方指明对象数量要比在两个地方指明好些。

### 13.3.1 使指针更像数组

作为题外话，上面定义的**fp**可以被修改指向任何类型，但这对于一个数组的起始地址来讲没有什么意义。一般来说，把它定义为常量会更好些，因为这样任何修改指针的企图都会被认为出错。为了得到这个效果，可以试着用下面的表达式：

```
int const* q = new int[10];
```

或

```
const int* q = new int[10];
```

上面的这两种表达式都把**const**和被指针指向的**int**捆绑在一起，而不是指针本身。如果使用下面的表达式：

```
int* const q = new int[10];
```

则现在**q**中的数组元素可以被修改了，但对**q**本身的修改（例如**q++**）是不合法的，因为它是一个普通数组标识符。

## 13.4 耗尽内存

当**operator new()**找不到足够大的连续内存块来安排对象时，将会发生什么事情呢？一个称为**new-handler**的特殊函数将会被调用。首先，检查指向函数的指针，如果指针非0，那么它指向的函数将被调用。

**new-handler**的默认动作是产生一个异常(*throw an exception*)，这个主题将在第2卷中介绍。然而，如果我们在程序里用堆分配，至少要用“内存已耗尽”的信息代替**new-handler**，并异常中断程序。用这个办法，在调试程序时会得到程序出了什么错误的线索。对于最终的程序，我们总想使之具有很强的容错恢复性。

通过包含**new.h**来替换**new-handler**，然后以想装入的函数地址为参数调用**set\_new\_handler()**函数。

```
//: C13:NewHandler.cpp
// Changing the new-handler
#include <iostream>
#include <cstdlib>
#include <new>
using namespace std;

int count = 0;

void out_of_memory() {
    cerr << "memory exhausted after " << count
        << " allocations!" << endl;
```

```

    exit(1);
}

int main() {
    set_new_handler(out_of_memory);
    while(1) {
        count++;
        new int[1000]; // Exhausts memory
    }
} ///:~

```

`new-handler` 函数必须不带参数且其返回值为`void`。`while`循环将持续分配`int`对象（并丢掉它们的返回地址）直到空的内存被耗尽。在紧接下去的下一次对`new`的调用时，将没有内存可被调用，所以调用`new-handler`。

`new-handler`的行为和`operator new()`绑在一起，如果已经重载了`operator new()`（在下一节中介绍），则`new-handle`将不会按默认调用。如果仍想调用`new-handler`，则我们不得不在重载了的`operator new()`的代码中加上做这些工作的代码。

当然，可以写更复杂的`new-handler`，甚至它可以回收内存〔通常叫做无用单元收集器 (*garbage collector*)〕。但这不是编程新手的工作。

### 13.5 重载`new`和`delete`

当我们创建一个`new`表达式时，会发生两件事。首先，使用`operator new()`分配内存，然后调用构造函数。在`delete`表达式里，调用了析构函数，然后使用`operator delete()`释放内存。我们无法控制构造函数和析构函数的调用（否则可能会意外地搅乱它们），但可以改变内存分配函数`operator new()`和`operator delete()`。

使用了`new`和`delete`的内存分配系统是为通用目的而设计的。但在特殊的情形下，它并不能满足需要。最常见的改变分配系统的原因是出于效率考虑：也许要创建和销毁一个特定的类的非常多的对象以至于这个运算变成了速度的瓶颈。C++允许重载`new`和`delete`来实现我们自己的存储分配方案，所以可以用它来处理问题。

另一个问题是堆碎片：分配不同大小的内存可能会在堆上产生很多碎片，以至于很快用完内存。虽然内存可能还有，但由于都是碎片，也就找不到足够大的内存块满足需要。通过为特定类创建自己的内存分配器，可以确保这种情况不会发生。

在嵌入和实时系统里，程序可能必须在有限的资源情况下运行很长时间。这样的系统也可能要求分配内存花费相同的时间且不允许出现堆内存耗尽或出现很多碎片的情况。由客户定制的内存分配器是一种解决办法，否则程序设计者在这种情况下要避免使用`new`和`delete`，而这将错过了C++很有价值的优点。

当重载`operator new()`和`operator delete()`时，我们只是改变了原有的内存分配方法，记住这一点是很重要的。编译器将用重载的`new`代替默认的版本去分配内存，然后为那个内存调用构造函数。所以，虽然当编译器看到`new`时，编译器分配内存并调用构造函数，但是当重载`new`时，可以改变的只是内存分配部分（`delete`也有相似的限制。）。

当重载`operator new()`时，也可以替换它用完内存时的行为，所以必须在`operator new()`里决定做什么：返回0、写一个调用`new-handler`的循环、再试着分配或者（典型的）产生一个`bad_alloc`的异常信息（在第2卷中讨论，可从[www.BruceEckel.com](http://www.BruceEckel.com)处获得）。

重载`new`和`delete`与重载任何其他运算符一样。但可以选择重载全局内存分配函数或者是针对特定类的分配函数。

### 13.5.1 重载全局`new`和`delete`

当全局版本的`new`和`delete`不能满足整个系统时，对其重载是很极端的方法。如果重载全局版本，就使默认版本完全不能被访问——甚至在这个重新定义里也不能调用它们。

重载的`new`必须有一个`size_t`参数（`size_t`的标准C类型）。这个参数由编译器产生并传递给我们，它是要分配内存的对象的长度。必须返回一个指向等于这个长度（或大于这个长度，如果有这样做的原因）的对象的指针，如果没有找到存储单元（在这种情况下，构造函数不被调用），则返回一个0。然而如果找不到存储单元，不能仅仅返回0，也许还应该做一些诸如调用`new-handler`或产生一个异常信息之类的事，通知这里存在问题。

`operator new()`的返回值是一个`void*`，而不是指向任何特定类型的指针。所做的是分配内存，而不是完成一个对象建立——直到构造函数调用了才完成对象的创建，它是编译器确保做的动作，不在我们的控制范围之内。

`operator delete()`的参数是一个指向由`operator new()`分配的内存的`void*`。参数是一个`void*`是因为它是在调用析构函数后得到的指针。析构函数从存储单元里移去对象。`operator delete()`的返回类型是`void`。

下面提供了一个如何重载全局`new`和`delete`的简单的例子：

```
//: C13:GlobalOperatorNew.cpp
// Overload global new/delete
#include <cstdio>
#include <cstdlib>
using namespace std;

void* operator new(size_t sz) {
    printf("operator new: %d Bytes\n", sz);
    void* m = malloc(sz);
    if(!m) puts("out of memory");
    return m;
}

void operator delete(void* m) {
    puts("operator delete");
    free(m);
}

class S {
    int i[100];
public:
    S() { puts("S::S()"); }
    ~S() { puts("S::~S()"); }
};

int main() {
    puts("creating & destroying an int");
    int* p = new int(47);
    delete p;
    puts("creating & destroying an s");
```



```

    S* s = new S;
    delete s;
    puts("creating & destroying S[3]");
    S* sa = new S[3];
    delete []sa;
} ///:~

```

这里可以看到重载**new**和**delete**的通常形式。这里的内存分配使用了标准C库函数**malloc()**和**free()**（可能默认的**new**和**delete**也使用这些函数）。并且,它们还打印出了有关正在做什么的信息。注意,这里使用**printf()**和**puts()**而不是**iostreams**。因此,当创建了一个**iostream**对象时（像全局的**cin**、**cout**和**cerr**），它们调用**new**去分配内存。用**printf()**不会进入死锁状态,因为它不调用**new**来初始化本身。

在**main()**里,创建内建数据类型对象以证明在这种情况下也调用重载的**new**和**delete**。然后创建一个类型**S**的单个对象,接着创建一个类型**S**的数组。对于这个数组,从所需要的字节数目中可以看到,额外的内存被分配用于存放它所包含对象的数量信息。在所有情况下,都使用了全局重载版本的**new**和**delete**。

### 13.5.2 对于一个类重载**new**和**delete**

为一个类重载**new**和**delete**时,尽管不必显式地使用**static**,但实际上仍是在创建**static**成员函数。它的语法也和重载任何其他运算符一样。当编译器看到使用**new**创建自己定义的类的对象时,它选择成员版本的**operator new()**而不是全局版本的**new()**。但全局版本的**new**和**delete**仍为所有其他类型对象使用（除非它们有自己的**new**和**delete**）。

在下面的例子里为类**Framis**创建了一个非常简单的内存分配系统。程序开始时在静态数据区域留出一块存储单元。这块内存被用来为**Framis**类型的对象分配存储空间。为了标明哪块存储单元已被使用,这里使用了一个字节（byte）数组,一个字节代表一块存储单元。

```

//: C13:Framis.cpp
// Local overloaded new & delete
#include <cstdint> // Size_t
#include <fstream>
#include <iostream>
#include <new>
using namespace std;
ofstream out("Framis.out");

class Framis {
    enum { sz = 10 };
    char c[sz]; // To take up space, not used
    static unsigned char pool[];
    static bool alloc_map[];
public:
    enum { psize = 100 }; // frami allowed
    Framis() { out << "Framis()\n"; }
    ~Framis() { out << "~Framis() ... "; }
    void* operator new(size_t) throw(bad_alloc);
    void operator delete(void*);
};

unsigned char Framis::pool[psize * sizeof(Framis)];
bool Framis::alloc_map[psize] = {false};

```



```

// Size is ignored -- assume a Framis object
void*
Framis::operator new(size_t) throw(bad_alloc) {
    for(int i = 0; i < psize; i++)
        if(!alloc_map[i]) {
            out << "using block " << i << " ... ";
            alloc_map[i] = true; // Mark it used
            return pool + (i * sizeof(Framis));
        }
    out << "out of memory" << endl;
    throw bad_alloc();
}

void Framis::operator delete(void* m) {
    if(!m) return; // Check for null pointer
    // Assume it was created in the pool
    // Calculate which block number it is:
    unsigned long block = (unsigned long)m
        - (unsigned long)pool;
    block /= sizeof(Framis);
    out << "freeing block " << block << endl;
    // Mark it free:
    alloc_map[block] = false;
}

int main() {
    Framis* f[Framis::psize];
    try {
        for(int i = 0; i < Framis::psize; i++)
            f[i] = new Framis;
        new Framis; // Out of memory
    } catch(bad_alloc) {
        cerr << "Out of memory!" << endl;
    }
    delete f[10];
    f[10] = 0;
    // Use released memory:
    Framis* x = new Framis;
    delete x;
    for(int j = 0; j < Framis::psize; j++)
        delete f[j]; // Delete f[10] OK
} ///:~

```

通过创建一个能够容纳`psize`个`Framis`对象的字节数组的方法，为`Framis`堆分配了内存。相应地，分配表中也会含有`psize`个成员，其中每一`bool`类型成员对应一块内存。初始化时，分配表中所有的值都被置为`false`，这可以使用设置首元素的聚合初始化技巧，因为编译器能够自动地以常规的预设值来初始化其余的元素（对于`bool`类型来说，就是初始化为`false`）。

局部`operator new()`和全局`operator new()`具有相同的语法。首先对分配表进行搜索，寻找值为`false`的成员。找到后将该成员设置为`true`，以此声明对应的存储单元已经被分配了，并且返回这个存储单元的地址。如果找不到任何空闲内存，将会给跟踪文件发送一个消息，并且产生一个`bad_alloc`类型的异常信息。

这是在这本书中看到的第一个含有异常情况的例子。因为有关异常情况的详细的讨论被

放在了第2卷，所以这里只是一个简单的例子。在**operator new()**中，可以看到两个异常情况处理的标志。首先是在函数参数表后面的**throw(bad\_alloc)**，它通知了编译器和读者这个函数可以产生一个**bad\_alloc**的异常信息。其次，如果没有内存可供使用了，则此函数会由**throw bad\_alloc**语句产生一个异常信息。当此异常信息产生时，函数停止执行并且把控制权交给表示为一个**catch**子句的异常处理 (*exception handler*)。

在**main()**中，可以看到异常处理的其余部分，也就是**try-catch**子句。被大括号围起的**Try**部分包含了可以产生异常信息的所有代码——在这里就是指任何对含有**Framis**对象的**new**的调用。跟在**try**部分后面的是一个或多个**catch**子句，每一个都指明了它们获取的异常信息的类型。在本例中，**catch(bad\_alloc)**指明了**bad\_alloc**类型的异常信息在此可被获取。这里的**catch**子句仅当**bad\_alloc**类型异常信息生成时才执行，并且执行是在这组**catch**子句的最后一个结束后开始的（这里只有一个**catch**子句，但在别的程序中可以有多个）。

在本例中，因为没有涉及全局**operator new()**和**delete()**，所以使用**iostreams**是可行的。

**operator delete()**假设**Framis**的地址是在这个堆里创建的，这是一个正确的假设。因为无论何时我们在堆上创建单个的**Framis**对象——不是一个数组，都将调用局部**operator new()**。而全局版本的**new()**在创建数组时使用。因此用户可能会在用**operator delete()**删除一个数组时，偶然地忘记了使用空方括号语法，而这就会出现错误。用户也可能删除了在栈上创建的指向对象的指针。如果考虑到这样的事情可能发生，应该加入一行代码以确保地址是在这个堆内并是在正确的地址范围内（也可以考虑重载**new**和**delete**对于防止内存丢失的潜力）。

**operator delete()**计算出该指针所代表的那块内存，并在分配表中将对应部分置为false，以表明这块内存已经被释放了。

在**main()**中，动态地分配足够多的**Framis**对象，把可用内存消耗掉。这用来检查无内存可供分配的情况。然后释放一个对象，再创建一个对象以表明释放的内存可被重新使用。

因为这个内存分配方案是针对**Framis**对象的，所以可能比使用默认的**new**和**delete**的通用内存分配方案效率要高一些。但是，应当注意，如果使用继承，该分配方案不能自动继承使用（继承将在第14章中介绍）。

### 13.5.3 为数组重载new和delete

如果为一个类重载了**operator new()**和**operator delete()**，那么无论何时创建这个类的一个对象都将调用这些运算符。但如果要创建这个类的一个对象数组时，全局**operator new()**就会被立即调用，用来为这个数组分配足够的内存。对此，可以通过为这个类重载运算符的数组版本，即**operator new[]**和**operator delete[]**，来控制对象数组的内存分配。下面的例子显示了何时这两个不同的版本会被调用：

```
//: C13:ArrayOperatorNew.cpp
// Operator new for arrays
#include <new> // Size_t definition
#include <fstream>
using namespace std;
ofstream trace("ArrayOperatorNew.out");

class Widget {
    enum { sz = 10 };
    int i[sz];
```

```

public:
    Widget() { trace << "Widget"; }
    ~Widget() { trace << "~Widget"; }
    void* operator new(size_t sz) {
        trace << "Widget::new: "
              << sz << " bytes" << endl;
        return ::new char[sz];
    }
    void operator delete(void* p) {
        trace << "Widget::delete" << endl;
        ::delete []p;
    }
    void* operator new[](size_t sz) {
        trace << "Widget::new[]: "
              << sz << " bytes" << endl;
        return ::new char[sz];
    }
    void operator delete[](void* p) {
        trace << "Widget::delete[]" << endl;
        ::delete []p;
    }
};

int main() {
    trace << "new Widget" << endl;
    Widget* w = new Widget;
    trace << "\ndelete Widget" << endl;
    delete w;
    trace << "\nnew Widget[25]" << endl;
    Widget* wa = new Widget[25];
    trace << "\ndelete []Widget" << endl;
    delete []wa;
} ///:~

```

这里，全局版本的**new**和**delete**被调用，除了加入了跟踪信息以外，它们和没有**new**和**delete**的重载版本效果是一样的。当然，可以在重载的**new**和**delete**里使用任意的内存分配方案。

可以看到，在语法上，除了多一对括号外，数组版本的**new**和**delete**与单个对象版本的是一样的。不管是哪种版本，我们都要决定所要分配内存的大小。数组版本中的大小指的是整个数组的大小。应该记住，重载**operator new()**惟一需要做的是返回一个足够大的内存块的指针。虽然可以初始化那块内存，但通常编译器将自动地调用构造函数来对该内存块进行初始化。

这里构造函数和析构函数只是打印出字符，因此可以看到什么时候它们被调用。下面是某个编译器生成的跟踪文件的输出信息：

```

new Widget
Widget::new: 40 bytes
*
delete Widget
~Widget::delete

new Widget[25]
Widget::new[]: 1004 bytes
*****
delete []Widget
~~~~~Widget::delete[]

```



正如所预计的, 创建单个对象需要40个字节 (本机为`int`类型分配4个字节)。首先调用 `operator new()`, 接着调用了构造函数 (这可从输出信息\*看出)。在后面的部分, 对`delete`的调用首先引起了析构函数的调用, 然后是对`operator delete()`的调用。

当创建一个`Widget`类型的对象数组时, 使用了数组版本的`operator new()`。但请注意, 需要的长度比期望的多了4个字节。这额外的4个字节是系统用来存放数组信息的, 特别是数组中对象的数量。当用下面的表达式时:

```
delete []Widget;
```

方括号就告诉编译器它是一个对象数组, 所以编译器产生寻找数组中对象的数量代码, 然后多次调用析构函数。可以看到, 即使数组`operator new()`和`operator delete()`只为整个数组调用一次, 但对于数组中的每一个对象, 都调用了默认的构造函数和析构函数。

### 13.5.4 构造函数调用

分析下面语句:

```
MyType* f = new MyType;
```

调用`new`分配了一个大小等于`MyType`类型的内存, 然后在那个内存上调用了`MyType`构造函数。但如果使用了`new`的内存分配没有成功, 将会出现什么状况呢? 在那种情况下, 构造函数不会被调用, 所以虽然没能成功地创建对象, 但至少没有调用构造函数并传给它一个为0的`this`指针。下面的例子说明了这一点:

```
//: C13:NoMemory.cpp
// Constructor isn't called if new fails
#include <iostream>
#include <new> // bad_alloc definition
using namespace std;

class NoMemory {
public:
    NoMemory() {
        cout << "NoMemory::NoMemory()" << endl;
    }
    void* operator new(size_t sz) throw(bad_alloc){
        cout << "NoMemory::operator new" << endl;
        throw bad_alloc(); // "Out of memory"
    }
};

int main() {
    NoMemory* nm = 0;
    try {
        nm = new NoMemory;
    } catch(bad_alloc) {
        cerr << "Out of memory exception" << endl;
    }
    cout << "nm = " << nm << endl;
} ///:~
```

当程序运行时, 并没有打印出构造函数的信息, 仅仅是打印了`operator new()`和异常处理的信息。因为`new`没有返回, 构造函数也没有被调用, 当然它的信息就不会被打印出来。

**nm**被初始化为0是很重要的, 因为**new**表达式没有执行完毕, 指针被置为0可以确保我们没有误用它。但是, 在异常处理中, 我们除了打印出一条信息以外, 还应当多做一些事情, 使得程序继续执行, 就像该对象已经被成功地创建了一样。理想情况下, 我们所做的将使程序从问题中恢复过来, 或者至少可以在记录下错误后退出。

在以前的C++版本中, 如果内存分配失败, 则一般是返回0。它将使构造函数不被调用。但是, 如果试着在一个标准的编译器中由**new**返回0值, 则会被告之应该产生一个**bad\_alloc**。

### 13.5.5 定位new和delete

重载**operator new()**还有其他两个不常见的用途。

- 1) 我们也许会想在内存的指定位置上放置一个对象。这对于面向硬件的内嵌系统特别重要, 在这个系统中, 一个对象可能和一个特定的硬件是同义的。
- 2) 我们也许会想在调用**new**时, 能够选择不同的内存分配方案。

这两个特性可以用相同的机制实现: 重载的**operator new()**可以带一个或多个参数。正如前面所看到的, 第一个参数总是对象的长度, 它在内部计算出来并由编译器传递给**new**。但其他参数可由我们自己定义: 一个放置对象的地址、一个是对内存分配函数或对象的引用, 或其他任何使我们方便的设置。

最初在调用过程中传递额外的参数给**operator new()**的方法看起来似乎有点古怪: 在关键字**new**后是参数表 (没有**size\_t**参数, 它由编译器处理), 参数表后面是正在创建的对象类名字。例如:

```
X* xp = new(a) X;
```

将**a**作为第二个参数传递给**operator new()**。当然, 这是在**operator new()**已经声明的情况下才是有效的。

下面的例子显示了如何在一个特定的存储单元里放置一个对象。

```
//: C13:PlacementOperatorNew.cpp
// Placement with operator new()
#include <cstdint> // Size_t
#include <iostream>
using namespace std;

class X {
    int i;
public:
    X(int ii = 0) : i(ii) {
        cout << "this = " << this << endl;
    }
    ~X() {
        cout << "X::~~X(): " << this << endl;
    }
    void* operator new(size_t, void* loc) {
        return loc;
    }
};

int main() {
    int l[10];
```



```

    cout << "l = " << l << endl;
    X* xp = new(l) X(47); // X at location l
    xp->X::~X(); // Explicit destructor call
    // ONLY use with placement!
} ///::~~

```

注意：**operator new()**仅返回了传递给它的指针。因此，调用者可以决定将对象存放在哪里，这时在该指针所指向的那块内存上，作为**new**表达式一部分的构造函数将被调用。

虽然本例只是使用了一个外加的参数，但如果需要实现其他的目的时，使用更多的参数同样也是可以的。

在销毁对象时将会出现两难选择的局面。因为仅有一个版本的**operator delete**，所以没有办法说“对这个对象使用我的特殊内存释放器”。可以调用析构函数，但不能用动态内存机制释放内存，因为内存不是在堆上分配的。

解决方法是用非常特殊的语法：我们可以显式地调用析构函数。例如：

```
xp->X::~X(); // Explicit destructor call
```

这里要严重警告一下。因为当某些人想要实时地决定对象的生存时间时，他们使用这种方法在作用范围结束之前的任意时刻销毁对象，而不是调节作用范围或者使用动态对象创建（这样做会更正确）。而如果用这种方法为在栈上创建的对象调用析构函数时，将会出现严重的问题，这是因为析构函数在对象超出作用范围时又会被调用一次。如果为在堆上创建的对象用这种方法调用析构函数，析构函数将被执行，但内存不释放，这是我们所不希望的。用这种方法显式地调用析构函数，其实只有一个原因，即支持**operator new()**的定位语法。

还有一个定位**operator delete**，它仅在一个定位**operator new**表达式的构造函数产生一个异常信息时才被调用（因此该内存在异常处理操作中被自动地清除了）。定位**operator delete**有一个和定位**operator new**相对应的参数表，该定位**operator new**是指在构造函数产生异常信息之前被调用的那一个。这个主题将放在第2卷的异常处理章节中。

## 13.6 小结

在栈上创建自动对象既方便又理想，但为了解决常见的程序问题，必须在程序执行的任何时候，特别是需要对来自程序外部信息作出反应时，能够创建和销毁对象。虽然C的动态内存分配可以从堆上得到内存，但它在C++上不易使用并且不能够保证安全。使用**new**和**delete**进行动态对象创建，这已经成为语言的核心，它可以使我们在堆上创建对象像在栈上创建对象一样容易。另外，它还增加了程序的灵活性。如果**new**和**delete**不能满足要求，尤其是它们的效率不高时，程序员可以改变它们的行为，而且在堆的内存用完时可以修改它们所执行的操作。

## 13.7 练习

部分练习题的答案可以在本书的电子文档“*Annotated Solution Guide for Thinking in C++*”中找到，只需支付很少的费用就可以从<http://www.BruceEckel.com>得到这个电子文档。

13-1 创建一个**class Counted**，它包含一个**int**类型的成员变量**id**和一个**static int**类型的成员变量**count**。默认构造函数的开头为**Counted() : id(count++)**。要求构造函数打印**id**值并且输出“it’s being created”。另外析构函数也打印出**id**值并且输出“it’s being destroyed”。

测试这个类。

- 13-2 通过使用**new**创建一个（练习 1 中的）**class Counted**的对象，并且用**delete**销毁它，来证明**new**和**delete**总是调用了构造函数和析构函数。在堆上创建和销毁这些对象的一个数组。
- 13-3 创建一个**PStash**对象，在此对象中用**new**创建练习 1 的对象。观察当这个对象超出了范围和它的析构函数被调用时有什么情况发生。
- 13-4 创建一个**vector< Counted\*>**，对它使用**new**创建（练习 1 中的）。**Counted**对象时返回的指针填充。扫描这个**vector**并输出**Counted**对象，然后再次扫描，并删除每一个对象。
- 13-5 重复练习 4 的操作，只是增加一个**Counted**的成员函数**f()**，该函数可以输出一条信息。然后扫描这个**vector**并对每一个对象调用函数**f()**。
- 13-6 使用**PStash**重复练习 5 的操作。
- 13-7 使用第 9 章的**Stack4.h**重复练习 5 的操作。
- 13-8 动态创建一个（练习 1 中的）**class Counted**的对象数组。不使用方括号对返回指针调用**delete**。对此操作的结果进行解释。
- 13-9 使用**new**创建一个（练习 1 中的）**class Counted**的对象，对**void\***类型的返回指针进行类型转换，然后再删除它。对此运算结果进行解释。
- 13-10 在计算机上执行**NewHandler.cpp**，观察变量**count**的最终结果。计算可供我们的程序使用的空闲内存的数量。
- 13-11 创建一个类，带有重载运算符**new**和**delete**，要求含有对于单个对象和数组的两个版本。演示这两个版本的工作情况。
- 13-12 设计一个对**Framis.cpp**进行测试的程序来显示定制的**new**和**delete**比全局的**new**和**delete**大约快多少。
- 13-13 修改**NoMemory.cpp**让它含有一个**int**类型的数组。使它实际上没有产生**bad\_alloc**，而是分配了内存。在**main()**中，建立一个像在**NewHandler.cpp**中的**while**循环，用来消耗完内存，观察一下当**operator new**没有测试内存是否被成功地分配时会有什么发生。然后在**operator new**中加入测试并产生**bad\_alloc**。
- 13-14 创建一个含有定位**new**运算符的类，定位**new**运算符的第二个参数是一个**string**类型值。这个类还包括一个**static vector<string>**，用来存放第二个参数。该定位**new**运算符同常规的一样用来分配内存。在**main()**中，调用定位**new**并且以描述该调用的字符串作为**string**参数（可能要用到预处理的**\_\_FILE\_\_**和**\_\_LINE\_\_**宏）。
- 13-15 通过增加**static vector<Widget\*>**来修改**ArrayOperatorNew.cpp**，即加入每一个**Widget**地址。该**Widget**地址是由**operator new()**分配内存并且当它被释放时可以通过**operator delete()**删除。（我们可能需要在标准 C++ 库文件或者在本书的第 2 卷（可从 Web 站点中获得）中查找有关**vector**的信息。）创建第二个类**MemoryChecker**，含有可以打印出**vector**中**Widget**指针数目的析构函数。再创建一个含有**MemoryChecker**对象的程序。在**main()**中，动态地分配且销毁一些**Widget**的对象和数组。显示**MemoryChecker**可以阻止内存丢失。

## 继承和组合

C++最重要的特征之一是代码重用。但是如果希望更进一步，就不能仅仅用拷贝代码和修改代码的方法，而是要做更多的工作。

在C中，这个问题未能得到很好的解决。而在C++中，这可以通过类的方法解决。我们通过创建新类来重用代码，而不是从头创建它们。这样，便可以使用别人已经创建好并经过调试的类。

关键技巧是使用这些类，但不修改已存在的代码。在本章中，我们将看到两种完成这项任务的方法。第一种方法很直接：我们简单地在新类中创建已存在类的对象。因为新类是由已存在类的对象组合而成，所以这种方法称为组合 (*composition*)。

第二种方法要复杂些。我们创建一个新类作为一个已存在类的类型。我们不修改已存在的类，而是采取这个已存在类的形式，并将代码加入其中。这种巧妙的方法称为继承 (*inheritance*)，其中大量的工作是由编译器完成。继承是面向对象程序设计的基石，而且它还有另外的含义，我们将在第15章中探讨它的另外含义。

在语法上和行为上，组合和继承大部分是相似的（它们都是在已存在类型的基础上创建新类型的方法）。在本章中，我们将学习这些代码重用机制。

### 14.1 组合语法

实际上，我们一直都在用组合创建类，只不过是在用内建数据类型（有时用**string**）组合新类。其实使用用户定义类型组合新类同样很容易。

考虑下面这个在某种意义上是有价值的类：

```
//: C14:Useful.h
// A class to reuse
#ifndef USEFUL_H
#define USEFUL_H

class X {
    int i;
public:
    X() { i = 0; }
    void set(int ii) { i = ii; }
    int read() const { return i; }
    int permute() { return i = i * 47; }
};
#endif // USEFUL_H ///:~
```

在X类中，数值成员是私有的，所以将类型X的一个对象作为公共对象嵌入到一个新类内部，这是绝对安全的。这样就使得新类的接口很简单，

```
//: C14:Composition.cpp
// Reuse code with composition
```

```
#include "Useful.h"

class Y {
    int i;
public:
    X x; // Embedded object
    Y() { i = 0; }
    void f(int ii) { i = ii; }
    int g() const { return i; }
};

int main() {
    Y y;
    y.f(47);
    y.x.set(37); // Access the embedded object
} ///:~
```

访问嵌入对象（称为子对象）的成员函数只需再一次的成员选择。

更常见的是把嵌入的对象设为私有，因此它们将成为内部实现的一部分（这意味着如果我们愿意，可以改变这个实现）。新类的公有接口函数包括了对嵌入对象的使用，但没有必要模仿这个对象的接口。

```
//: C14:Composition2.cpp
// Private embedded objects
#include "Useful.h"
class Y {
    int i;
    X x; // Embedded object
public:
    Y() { i = 0; }
    void f(int ii) { i = ii; x.set(ii); }
    int g() const { return i * x.read(); }
    void permute() { x.permute(); }
};

int main() {
    Y y;
    y.f(47);
    y.permute();
} ///:~
```

这里，**permute()**函数是通过新类的接口执行的，但**X**的其他的成员函数是在新类的成员**Y**内执行的。

## 14.2 继承语法

组合的语法是清晰的，而对于继承，则有新的不同的形式。

当继承时，我们会发现“这个新类很像原来的类”。我们规定，在代码中和原来一样给出该类的名字，但在类的左括号的前面，加一个冒号和基类的名字（对于多重继承，要给出多个基类名，它们之间用逗号分开）。当做完这些时，将会自动地得到基类中的所用数据成员和成员函数。下面是一个例子：

```
//: C14:Inheritance.cpp
```

```

// Simple inheritance
#include "Useful.h"
#include <iostream>
using namespace std;

class Y : public X {
    int i; // Different from X's i
public:
    Y() { i = 0; }
    int change() {
        i = permute(); // Different name call
        return i;
    }
    void set(int ii) {
        i = ii;
        X::set(ii); // Same-name function call
    }
};

int main() {
    cout << "sizeof(X) = " << sizeof(X) << endl;
    cout << "sizeof(Y) = "
        << sizeof(Y) << endl;
    Y D;
    D.change();
    // X function interface comes through:
    D.read();
    D.permute();
    // Redefined functions hide base versions:
    D.set(12);
} ///:~

```

我们可以看到Y对X进行了继承，这意味着Y将包含X中的所有数据成员和成员函数。实际上，正如没有对X进行继承，而在Y中创建了一个X的成员对象一样，Y是包含了X的一个子对象。无论是成员对象还是基类存储，都被认为是子对象。

所有X中的私有成员在Y中仍然是私有的，这是因为Y对X进行了继承并不意味着Y可以不遵守保护机制。X中的私有成员仍然占有存储空间，只是不可以直接地访问它们罢了。

在main()中，从sizeof(Y)是sizeof(X)的两倍可以看出，Y的数据成员是同X的成员结合在一起了。

我们注意到，本例中的基类前面是public。由于在继承时，基类中所有的成员都是被预设为主有的，所以如果基类的前面没有public，这意味着基类的所有公有成员将在派生类中变为私有的。这显然不是所希望的<sup>①</sup>，我们希望基类中的所有公有成员在派生类中仍是公有的。这可以在继承时通过使用关键字public来实现。

在change()中，基类的permute()函数被调用。即派生类可以直接访问所有基类的公有函数。

派生类中的set()函数重新定义了基类中set()函数。这即是说，如果调用一个Y类型对象的read()和permute()函数，将会使用基类中的这些函数（这可在main()中表现出来）。但如果调用一个Y类型对象的set()函数，将会使用派生类中的重定义版本。这意味着如果不想使用某个继承而来的函数，我们可以改变它的内容（当然我们也可以增加全新的函数，例如

① 在Java中，编译器不会因继承而让程序员减少对成员的访问能力。

`change()`。

然而，当我们重新定义了一个函数的后，仍可能想调用基类的函数。但如果对于 `set()`，只是简单地调用 `set()` 函数，将得到这个函数的本地版本——一个递归的函数调用。为了调用基类的 `set()` 函数，必须使用作用域运算符来显式地标明基类名。

### 14.3 构造函数的初始化表达式表

已经看到，在C++中保证正确的初始化是多么重要，这一点在组合和继承中也是一样。当创建一个对象时，编译器确保调用了所有子对象的构造函数。到目前为止，在已有的例子中，所有子对象都有默认的构造函数，编译器可以自动调用它们。但是，如果子对象没有默认构造函数或如果想改变构造函数的某个默认参数，情况怎么样呢？这会出现问题的，因为这个新类的构造函数没有权利访问这个子对象的私有数据成员，所以不能直接地对它们初始化。

解决的方法很简单：对于子对象调用构造函数，C++为此提供了专门的语法，即构造函数的初始化表达式表。构造函数的初始化表达式表的形式模仿继承活动。对于继承，我们把基类置于冒号和这个类体的左括号之间。而在构造函数的初始化表达式表中，可以将对子对象构造函数的调用语句放在构造函数参数表和冒号之后，在函数体的左括号之前。对于从 **Bar** 继承来的类 **MyType**，如果 **Bar** 的构造函数只有一个 `int` 型参数，则可以表示为：

```
MyType::MyType(int i) : Bar(i) { // ...
```

#### 14.3.1 成员对象初始化

显然，对于组合，也可以对成员对象使用同样语法，只是所给出的不是类名，而是对象的名字。如果在初始化表达式表中有多个构造函数的调用，应当用逗号加以隔开：

```
MyType2::MyType2(int i) : Bar(i), m(i+1) { // ...
```

这是类 **MyType2** 构造函数的开头，该类是从 **Bar** 继承来的，并且包含一个称为 **m** 的成员对象。请注意，虽然可以在这个构造函数的初始化表达式表中看到基类的类型，但只能看到成员对象的标识符。

#### 14.3.2 在初始化表达式表中的内建类型

构造函数的初始化表达式表允许我们显式地调用成员对象的构造函数。事实上，也没有其他方法可以调用那些构造函数。它的主要思想是，在进入新类的构造函数体之前调用所有其他的构造函数。这样，对子对象的成员函数所做的任何调用都总是转到了这个被初始化的对象中。即使编译器可以隐藏地调用默认的构造函数，但在没有对所有的成员对象和基类对象的构造函数进行调用之前，就没有办法进入该构造函数体。这是C++的一个强化的机制，它确保了，如果没有调用对象（或对象的一部分）的构造函数，就别想向下进行。

所有的成员对象在构造函数的左括号之前就被初始化了，这种方法对于程序设计很有帮助。一旦遇到左括号，就认为所有的子对象已被正确地初始化了，我们的精力就可以集中在想要完成的任务上面。然而，还有一个问题：对于那些没有构造函数的内建类型嵌入对象，这一切又将怎样呢？

为了使语法一致，可以把内建类型看做这样一种类型，它只有一个取单个参数的构造函



数，而这个参数与正在初始化的变量的类型相同。于是，可以这样写：

```
//: C14:PseudoConstructor.cpp
class X {
    int i;
    float f;
    char c;
    char* s;
public:
    X() : i(7), f(1.4), c('x'), s("howdy") {}
};

int main() {
    X x;
    int i(100); // Applied to ordinary definition
    int* ip = new int(47);
} ///:~
```

这些“伪构造函数调用”操作可以进行简单的赋值。这种方法很方便，并且具有良好的编码风格，所以常能看到它使用。

甚至当在类之外创建内建类型的变量时，也可以使用伪构造函数语法：

```
int i(100);
int* ip = new int(47);
```

这使得内建类型的操作有点类似于对象。但要记住，这些并不是真正的构造函数。特别地，如果没有显式的进行伪构造函数调用，初始化是不会执行的。

## 14.4 组合和继承的联合

当然，还可以把组合和继承放在一起使用。下面的例子中通过继承和组合两种方法创建了一个更复杂的类。

```
//: C14:Combined.cpp
// Inheritance & composition

class A {
    int i;
public:
    A(int ii) : i(ii) {}
    ~A() {}
    void f() const {}
};

class B {
    int i;
public:
    B(int ii) : i(ii) {}
    ~B() {}
    void f() const {}
};

class C : public B {
    A a;
public:
```



```

C(int ii) : B(ii), a(ii) {}
~C() {} // Calls ~A() and ~B()
void f() const { // Redefinition
    a.f();
    B::f();
}
};

int main() {
    C c(47);
} ///:~

```

**C**对**B**进行了继承并且有一个类型**A**的成员对象（“由类型**A**的成员对象组合而成”）。可以看到，构造函数的初始化表达式表中调用了基类构造函数和成员对象构造函数。

函数 **C::f()** 重定义了它所继承的 **B::f()**，但同时还调用基类版本。另外，它还调用了 **a.f()**。注意，只有通过继承，才能重新定义它的函数。而对于成员对象，只能操作这个对象的公共接口，而不能重定义它。另外，如果 **C::f()** 还没有被定义，则对类型 **C** 的一个对象调用 **f()** 就不会调用 **a.f()**，而会调用 **B::f()**。

#### 自动析构函数调用

虽然常常需要在初始化表达式表中做显式构造函数调用，但并不需要做显式的析构函数调用，因为对于任何类型只有一个析构函数，并且它并不取任何参数。然而，编译器仍要保证所有的析构函数被调用，这意味着，在整个层次中的所有析构函数中，从派生最底层的析构函数开始调用，一直到根层。

值得强调的是，在每一层中构造函数和析构函数都被调用的情况相当罕见。然而对于通常的成员函数，只是这个函数被调用，而它的那些基类版本并不会被调用。如果还想调用重载过的成员函数的基类版本，则必须显式地去做。

### 14.4.1 构造函数和析构函数调用的次序

当一个对象有许多子对象时，了解构造函数和析构函数的调用次序是很有意思的。下面的例子清楚地表明了调用的次序：

```

//: C14:Order.cpp
// Constructor/destructor order
#include <fstream>
using namespace std;
ofstream out("order.out");

#define CLASS(ID) class ID { \
public: \
    ID(int) { out << #ID " constructor\n"; } \
    ~ID() { out << #ID " destructor\n"; } \
};

CLASS(Basel);
CLASS(Member1);
CLASS(Member2);
CLASS(Member3);
CLASS(Member4);

class Derived1 : public Basel {

```



```

    Member1 m1;
    Member2 m2;
public:
    Derived1(int) : m2(1), m1(2), Base1(3) {
        out << "Derived1 constructor\n";
    }
    ~Derived1() {
        out << "Derived1 destructor\n";
    }
};

class Derived2 : public Derived1 {
    Member3 m3;
    Member4 m4;
public:
    Derived2() : m3(1), Derived1(2), m4(3) {
        out << "Derived2 constructor\n";
    }
    ~Derived2() {
        out << "Derived2 destructor\n";
    }
};

int main() {
    Derived2 d2;
} ///:~

```

首先, 创建一个**ofstream**对象, 用来把所有的输出发送到一个文件中。然后为了在书中少敲一些字符也为了演示一种宏技术 (这个技术将在第16章中被一个更好的技术代替), 这里使用了宏以建立一些类 (这些类将被用于继承和组合)。每个构造函数和析构函数向这个跟踪文件报告它们自己的行动。注意, 这些是构造函数, 而不是默认构造函数, 它们每一个都有一整型参数。这个参数本身没有标识符, 它的惟一的任務就是强迫在初始化表达式表中显式调用这些构造函数 (消除标识符防止编译器警告信息)。

这个程序的输出是:

```

Base1 constructor
Member1 constructor
Member2 constructor
Derived1 constructor
Member3 constructor
Member4 constructor
Derived2 constructor
Derived2 destructor
Member4 destructor
Member3 destructor
Derived1 destructor
Member2 destructor
Member1 destructor
Base1 destructor

```

可以看出, 构造是从类层次的最根处开始, 而在每一层, 首先会调用基类构造函数, 然后调用成员对象构造函数。调用析构函数则严格按照构造函数相反的次序——这是很重要的, 因为要考虑潜在的相关性 (对于派生类中的构造函数和析构函数, 必须假设基类子对象仍然可

供使用,并且已经被构造了——或者还未被消除)。

另一个有趣现象是,对于成员对象,构造函数调用的次序完全不受构造函数的初始化表达式表中的次序影响。该次序是由成员对象在类中声明的次序所决定的。如果能通过构造函数的初始化表达式表改变构造函数调用次序,那么就会对两个不同的构造函数有两种不同的调用顺序。而析构函数将不能知道如何相应逆序地执行析构,这就产生了相关性问题。

## 14.5 名字隐藏

如果继承一个类并且对它的成员函数重新进行定义,可能会出现两种情况。第一种是正如在基类中所进行的定义一样,在派生类的定义中明确地定义操作和返回类型。这称之为对普通成员函数的重定义 (*redefining*),而如果基类的成员函数是虚函数的情况,又可称之为重写 (*overriding*) (虚函数是一种常见的函数,我们将在第15章详细地进行介绍)。但是如果在派生类中改变了成员函数参数列表和返回类型,会发生什么情况呢? 这里有一个例子:

```

//: C14:NameHiding.cpp
// Hiding overloaded names during inheritance
#include <iostream>
#include <string>
using namespace std;

class Base {
public:
    int f() const {
        cout << "Base::f()\n";
        return 1;
    }
    int f(string) const { return 1; }
    void g() {}
};

class Derived1 : public Base {
public:
    void g() const {}
};

class Derived2 : public Base {
public:
    // Redefinition:
    int f() const {
        cout << "Derived2::f()\n";
        return 2;
    }
};

class Derived3 : public Base {
public:
    // Change return type:
    void f() const { cout << "Derived3::f()\n"; }
};

class Derived4 : public Base {

```



```

public:
    // Change argument list:
    int f(int) const {
        cout << "Derived4::f()\n";
        return 4;
    }
};

int main() {
    string s("hello");
    Derived1 d1;
    int x = d1.f();
    d1.f(s);
    Derived2 d2;
    x = d2.f();
    //! d2.f(s); // string version hidden
    Derived3 d3;
    //! x = d3.f(); // return int version hidden
    Derived4 d4;
    //! x = d4.f(); // f() version hidden
    x = d4.f(1);
} ///:~

```

在Base类中有一个可被重载的函数f(), 类Derived1并没有对函数f()进行任何改变, 但它重新定义了函数g()。在main()中, 可以看到函数f()的两个重载版本在类Derived1中都是可以使用的。但是, 由于类Derived2重新定义了函数f()的一个版本, 而对另一个版本没有进行重定义, 因此这第二个重载形式是不可以使用的。在类Derived3中, 通过改变返回类型隐藏了基类中的两个函数版本, 而在类Derived4中, 通过改变参数列表同样隐藏了基类中的两个函数版本。总体上, 可以得出, 任何时候重新定义了基类中的一个重载函数, 在新类之中所有其他的版本则被自动地隐藏了。在第15章, 我们将会看到加上virtual这个关键字会对函数的重载有一点影响。

如果通过修改基类中一个成员函数的操作与/或返回类型来改变了基类的接口, 我们就没有使用继承通常所提供的功能, 而是按另一种方式来重用了该类。这并不一定意味做错了, 只是由于继承的最终目标是为了实现多态性 (polymorphism)。并且如果我们改变了函数特征或返回类型, 实际上便改变了基类的接口。如果这便是想要做的, 我们就主要通过继承来重用代码, 而无须维护基类的通用接口 (这是多态性的一个很重要的方面)。总体上说, 当按这种方式使用继承, 就意味着我们有一个具有通用目的的类, 对于特定的需要再对它进行具体化——虽然不总是这样, 但通常这被认为是属于组合的范围。

例如, 对于第9章中的Stack类, 该类的一个问题是我们不得不在每次从容器中取回指针时进行类型变换。这不仅仅很麻烦, 而且不安全。我们可以把指针类型转换为指向想要的任何对象上。

初看较好的解决方法是通过继承的方法来具体化通用类Stack。下面的例子使用了第9章中的类:

```

//: C14:InheritStack.cpp
// Specializing the Stack class
#include "../C09/Stack4.h"
#include "../require.h"
#include <iostream>

```

```

#include <fstream>
#include <string>
using namespace std;

class StringStack : public Stack {
public:
    void push(string* str) {
        Stack::push(str);
    }
    string* peek() const {
        return (string*)Stack::peek();
    }
    string* pop() {
        return (string*)Stack::pop();
    }
    ~StringStack() {
        string* top = pop();
        while(top) {
            delete top;
            top = pop();
        }
    }
};

int main() {
    ifstream in("InheritStack.cpp");
    assure(in, "InheritStack.cpp");
    string line;
    StringStack textlines;
    while(getline(in, line))
        textlines.push(new string(line));
    string* s;
    while((s = textlines.pop()) != 0) { // No cast!
        cout << *s << endl;
        delete s;
    }
} //::~~

```

因为所有**Stack4.h**的成员函数都是内联的，所以不再需要进行链接。

**StringStack**类具体化了**Stack**类，所以**push()**将仅接收**String**类型指针。以前，**Stack**类接收**void**类型指针，但用户没有进行类型检查以确保插入了正确的指针。另外，**peek()**和**pop()**现在返回了**String**类型指针，而不是**void**类型指针，所以使用这些指针就无须进行类型转换了。

很不可思议，在**push()**、**peek()**和**pop()**中不需要额外的类型安全检查！在编译的时候，编译器会收到额外的类型信息，但这些函数是内联的并且不会产生额外的代码。

名字隐藏在这里起了作用，这主要是因为**push()**函数有着不同的特征：参数列表是不同的。如果在同一个类中含有两个版本的**push()**，它们将会被重载，但在这里并不希望进行重载，这是由于它仍将允许我们把任何类型的指针作为**void\***类型传送给**push()**。幸运的是，当**push()**函数的新版本在派生类中被定义后，C++将把基类中的**push(void\*)**版本隐藏起来，因此这时仅允许向**StringStack**中推入**push()** **string**指针。

由于现在可以确保我们清楚地知道在容器中的是何种类型的对象，所以析构函数能正

确地执行，同时对象的所属问题也就被解决了——或者说至少是解决对象所属问题的一种方法。这里，如果`push()`一个`string`指针进入`StringStack`，然后（根据`StringStack`的语义）我们也可以传递该指针的所属类给`StringStack`。如果`pop()`这个指针，将不仅可以得到该指针，再且还可以得到该指针的所属类。当调用析构函数时，任何留在`StringStack`中的指针将会被其析构函数消除。由于这里的都是`string`类型的指针，所以`delete`语句将作用于`string`指针，而不会对`void`指针进行操作，于是正确地执行了析构操作，这时一切都正确地运行。

这里有一个缺点：就是这个类仅仅可对`string`指针进行操作。如果想要一个可对某一其他类型的对象进行操作的`Stack`类，我们必须写一个该类的新版本，而它也仅可作用于这个新类型的对象，这将变得很麻烦。我们可在第16章中看到，这个问题最终将通过模板解决。

我们可以对这个例子多做一些观察：在继承的过程中，它改变了`Stack`类的接口。如果接口是不同的，一个`StringStack`类将不同于`Stack`类，我们也不能把`StringStack`类当做`Stack`类进行使用。这里充分表露了继承的不可靠性，如果我们不创建一个`Stack`类型的`StringStack`，那为什么要继承呢？更为恰当的`StringStack`版本将在本章后面给出。

## 14.6 非自动继承的函数

不是所有的函数都能自动地从基类继承到派生类中的。构造函数和析构函数用来处理对象的创建和析构操作，但它们只知道对它们的特定层次上的对象做些什么。所以，在该类以下各个层次中的所有的构造函数和析构函数都必须被调用，也就是说，构造函数和析构函数不能被继承，必须为每一个特定的派生类分别创建。

另外，`operator=` 也不能被继承，因为它完成类似于构造函数的活动。这就是说，尽管我们知道如何由等号右边的对象初始化左边的对象的所有成员，但这并不意味着这个初始化在继承后仍然具有同样的意义。

在继承过程中，如果不亲自创建这些函数，编译器就会生成它们（至于构造函数，我们不能创建任何的构造函数，因为编译器创建默认的构造函数和拷贝构造函数），这在第6章中已经简要地讲过了。被生成的构造函数使用成员方式的初始化，而被生成的 `operator=` 使用成员方式的赋值。下面是由编译器创建的函数的例子。

```
//: C14:SynthesizedFunctions.cpp
// Functions that are synthesized by the compiler
#include <iostream>
using namespace std;

class GameBoard {
public:
    GameBoard() { cout << "GameBoard()\n"; }
    GameBoard(const GameBoard&) {
        cout << "GameBoard(const GameBoard&)\n";
    }
    GameBoard& operator=(const GameBoard&) {
        cout << "GameBoard::operator=()\n";
        return *this;
    }
    ~GameBoard() { cout << "~GameBoard()\n"; }
};
```



```

class Game {
    GameBoard gb; // Composition
public:
    // Default GameBoard constructor called:
    Game() { cout << "Game()\n"; }
    // You must explicitly call the GameBoard
    // copy-constructor or the default constructor
    // is automatically called instead:
    Game(const Game& g) : gb(g.gb) {
        cout << "Game(const Game&)\n";
    }
    Game(int) { cout << "Game(int)\n"; }
    Game& operator=(const Game& g) {
        // You must explicitly call the GameBoard
        // assignment operator or no assignment at
        // all happens for gb!
        gb = g.gb;
        cout << "Game::operator=()\n";
        return *this;
    }
    class Other {}; // Nested class
    // Automatic type conversion:
    operator Other() const {
        cout << "Game::operator Other()\n";
        return Other();
    }
    ~Game() { cout << "~Game()\n"; }
};

class Chess : public Game {};

void f(Game::Other) {}

class Checkers : public Game {
public:
    // Default base-class constructor called:
    Checkers() { cout << "Checkers()\n"; }
    // You must explicitly call the base-class
    // copy constructor or the default constructor
    // will be automatically called instead:
    Checkers(const Checkers& c) : Game(c) {
        cout << "Checkers(const Checkers& c)\n";
    }
    Checkers& operator=(const Checkers& c) {
        // You must explicitly call the base-class
        // version of operator=() or no base-class
        // assignment will happen:
        Game::operator=(c);
        cout << "Checkers::operator=()\n";
        return *this;
    }
};

int main() {
    Chess d1; // Default constructor

```





```

    Chess d2(d1); // Copy-constructor
    //! Chess d3(1); // Error: no int constructor
    d1 = d2; // Operator= synthesized
    f(d1); // Type-conversion IS inherited
    Game::Other go;
    //! d1 = go; // Operator= not synthesized
        // for differing types
    Checkers c1, c2(c1);
    c1 = c2;
} ///:~

```

**GameBoard** 和 **Game** 中的构造函数和 **operator=** 都自己作了声明，所以我们能知道编译器何时使用它们。另外，**operator Other()** 从 **Game** 对象到被嵌入的类 **Other** 的对象完成自动类型变换。类 **Chess** 简单地从 **Game** 继承，并没有创建函数（观察编译器如何反应）。函数 **f()** 接收一个 **Other** 对象以测试这个自动类型变换函数。

在 **main()** 中，调用了为派生类 **Class** 生成的默认构造函数和拷贝构造函数。调用这些构造函数的 **Game** 版本作为构造函数调用继承的一部分，尽管这看上去像是继承，但新的构造函数实际上是创建的。正如所预料的，自动创建带参数的构造函数是不可能的，因为这样对于编译器来说需要靠直觉知道太多东西。

在 **Chess** 中，使用成员函数赋值，**operator=** 也被作为一个新的函数生成，（因此，调用了基类版本），这是因为该函数在新类中没有被显式地写出。当然，析构函数也会被编译器自动地生成。

鉴于有关处理对象创建的重写函数的所有原则，我们也许会觉得奇怪，为什么自动类型变换运算也能被继承。但其实这不足为奇——如果在 **Game** 中有足够的块建立一个 **Other** 对象，那么在从 **Game** 中派生出的任何东西中，这些块仍在原地，类型变换当然也就仍然有效（尽管实际上我们可能想重定义它）。

生成的 **operator=** 仅仅作用于同种类型对象。如果想把一种类型赋予另一种类型，则这个 **operator=** 必须由自己写出。

如果仔细地观察 **Game**，将会看到拷贝构造函数和赋值运算符显式地调用了成员对象的拷贝构造函数和赋值运算符。我们通常会想这么做的，因为如果不这样做的话，将会代替拷贝构造函数调用默认的成员对象构造函数，至于赋值运算符，则根本就不会对成员对象有赋值操作执行。

最后，观察一下 **Checkers**，它显示地写了默认构造函数、拷贝构造函数和赋值运算符。在默认构造函数中，默认的基类构造函数被自动地调用，这正是我们所希望的。但是，有一点很重要，一旦决定写自己的拷贝构造函数和赋值运算符，编译器就会假定我们已知道所做的一切，并且不再像在生成的函数中那样自动地调用基类版本。而如果想调用基类版本，那我们就必须亲自显式地调用它们。**Checkers** 的拷贝构造函数中，这个调用出现在构造函数的初始化列表中：

```
Checkers(const Checkers& c) : Game(c) {
```

至于 **Checkers** 赋值运算符，基类的调用在函数体的第一行中：

```
Game::operator=(c);
```

无论何时我们继承了一个类，这些调用都将成为我们使用的规范形式的一部分。

### 14.6.1 继承和静态成员函数

静态 (**static**) 成员函数与非静态成员函数的共同点:

- 1) 它们均可被继承到派生类中。
  - 2) 如果我们重新定义了一个静态成员, 所有在基类中的其他重载函数会被隐藏。
  - 3) 如果我们改变了基类中一个函数的特征, 所有使用该函数名字的基类版本都将会被隐藏。
- 然而, 静态 (**static**) 成员函数不可以是虚函数 (**virtual**) (第15章将详细介绍这个主题)。

## 14.7 组合与继承的选择

无论组合还是继承都能把子对象放在新类型中。两者都使用构造函数的初始化表达式表去构造这些子对象。现在我们可能会奇怪, 这两者之间到底有什么不同? 该如何选择?

组合通常是在希望新类内部具有已存在类的功能时使用, 而不是希望已存在类作为它的接口。这就是说, 嵌入一个对象用以实现新类的功能, 而新类的用户看到的是新定义的接口而不是来自老类的接口。为此, 在新类的内部嵌入已存在类的 **private** 对象。

有时, 又希望允许类用户直接访问新类的组成, 这就让成员对象是 **public**。由于成员对象使用自己的访问控制, 所以是安全的, 而当用户了解了我们所做的组装工作时, 会更容易理解接口。**Car**类是一个很好的例子:

```
//: C14:Car.cpp
// Public composition

class Engine {
public:
    void start() const {}
    void rev() const {}
    void stop() const {}
};

class Wheel {
public:
    void inflate(int psi) const {}
};

class Window {
public:
    void rollup() const {}
    void rolldown() const {}
};

class Door {
public:
    Window window;
    void open() const {}
    void close() const {}
};

class Car {
public:
    Engine engine;
    Wheel wheel[4];
};
```



```

    Door left, right; // 2-door
};

int main() {
    Car car;
    car.left.window.rollup();
    car.wheel[0].inflate(72);
} ///:~

```

因为**Car**的组合是分析这个问题的一部分（并不是基本设计的部分），所以让成员是**public**，有助于客户程序员理解如何使用这个类，而且能使类的实例具有更小的代码复杂性。

稍加思考就会看到，用“车辆”对象组合一个**Car**是毫无意义的——小汽车不能包含车辆，它本身就是一种车辆。这种*is-a*关系用继承表达，而*has-a*关系用组合表达。

### 14.7.1 子类型设置

现在假设想创建 **ifstream** 对象的一个类，它不仅打开一个文件，而且还保存文件名。这时可以使用组合并把 **ifstream** 及 **string** 都嵌入这个新类中：

```

//: C14:FName1.cpp
// An ifstream with a file name
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class FName1 {
    ifstream file;
    string fileName;
    bool named;
public:
    FName1() : named(false) {}
    FName1(const string& fname)
        : fileName(fname), file(fname.c_str()) {
        assure(file, fileName);
        named = true;
    }
    string name() const { return fileName; }
    void name(const string& newName) {
        if(named) return; // Don't overwrite
        fileName = newName;
        named = true;
    }
    operator ifstream&() { return file; }
};

int main() {
    FName1 file("FName1.cpp");
    cout << file.name() << endl;
    // Error: close() not a member;
    //! file.close();
} ///:~

```



然而这里存在一个这样的问题：我们也许想通过包含一个从 **FName1** 到 **ifstream** & 的自动类型转换运算，在任何使用 **ifstream** 的地方都使用 **FName1** 对象，但在 **main** 中，

```
file.close();
```

这一行将不编译，因为自动类型转换只发生在函数调用中，而不在成员选择期间。所以，这个方法不可行。

第二个方法是对 **FName1** 增加 **close()** 的定义：

```
void close() { file.close(); }
```

如果只有很少的函数要从 **ifstream** 类中拿来，这是可行的。在这种情况下，我们只是用了这个类的一部分，并且组合是适用的。

但是，如果希望这个类中的每件东西都进来，应该做什么呢？这称为子类型化 (*subtyping*)，因为我们正由已存在的类创建一个新类，并且希望这个新类与已存在的类有着严格相同的接口（希望增加任何我们想要加入的其他成员函数），所以能在已经用过这个已存在类的任何地方使用这个新类，这就是必须使用继承的地方。可以看到，子类型设置很好地解决了先前例子中的问题。

```
//: C14:FName2.cpp
// Subtyping solves the problem
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class FName2 : public ifstream {
    string fileName;
    bool named;
public:
    FName2() : named(false) {}
    FName2(const string& fname)
        : ifstream(fname.c_str()), fileName(fname) {
        assure(*this, fileName);
        named = true;
    }
    string name() const { return fileName; }
    void name(const string& newName) {
        if(named) return; // Don't overwrite
        fileName = newName;
        named = true;
    }
};

int main() {
    FName2 file("FName2.cpp");
    assure(file, "FName2.cpp");
    cout << "name: " << file.name() << endl;
    string s;
    getline(file, s); // These work too!
    file.seekg(-200, ios::end);
    file.close();
} ///:~
```



现在，能与`ifstream`对象一起工作的任何成员函数也能与`FName2`对象一起工作。我们也可以看见，需要使用`ifstream`对象的非成员函数，例如`getline()`，同样可以使用`FName2`对象。这是因为，一个`FName2`是`ifstream`的一个类型。它并不只是简单地包含了一个`ifstream`。这一点非常重要，我们将在本章最后和在第15章中进行讨论。

### 14.7.2 私有继承

通过在基类表中去掉 `public` 或者通过显式地声明`private`，可以私有地继承基类（后者可能是更好的策略，因为可以让用户明白它的含义）。当私有继承时，我们是“照此实现”；也就是说，创建的新类具有基类的所有数据和功能，但这些功能是隐藏的，所以它只是部分的内部实现。该类的用户访问不到这些内部功能，并且一个对象不能被看做是这个基类的实例（正如在`FName2.Cpp`中的）。

我们可能奇怪，`private` 继承的目的是什么，因为在这个新类中使用组合创建一个`private`对象的选择似乎更合适。为了完整性，`private`继承被包含在该语言中。但是，如果不为了其他理由，则应当减少混淆，所以通常希望使用组合而不是`private`继承。然而，这里可能偶然有这种情况，即可能想产生像基类接口一样的接口部分，而不允许该对象的处理像一个基类对象。`private`继承提供了这个功能。

#### 14.7.2.1 对私有继承成员公有化

当私有继承时，基类的所有`public`成员都变成了`private`。如果希望它们中的任何一个是可视的，只要用派生类的`public`部分声明它们的名字即可：

```
//: C14:PrivateInheritance.cpp
class Pet {
public:
    char eat() const { return 'a'; }
    int speak() const { return 2; }
    float sleep() const { return 3.0; }
    float sleep(int) const { return 4.0; }
};

class Goldfish : Pet { // Private inheritance
public:
    using Pet::eat; // Name publicizes member
    using Pet::sleep; // Both members exposed
};

int main() {
    Goldfish bob;
    bob.eat();
    bob.sleep();
    bob.sleep(1);
    //! bob.speak(); // Error: private member function
} ///:~
```

这样，如果想要隐藏基类的部分功能，则`private`继承是有用的。

注意给出一个重载函数的名字将使基类中所有它的重载版本公有化。

在使用`private`继承取代组合之前，应当仔细考虑，当与运行时类型标识相连时，私有继承特别复杂（这是本书第2卷中一章的主题，可从[www.BruceEckel.com](http://www.BruceEckel.com)处下载）。

## 14.8 protected

我们已经学习了继承，而关键字**protected**对于继承有特殊的意义。在理想情况下，**private**成员总是严格私有的，但在实际项目中，有时希望某些东西隐藏起来，但仍允许其派生类的成员访问。于是关键字**protected**派上了用场。它的意思是：“就这个类的用户而言，它是**private**的，但它可被从这个类继承来的任何类使用”。

最好让数据成员是**private**，因为我们应该保留改变内部实现的权利。然后才能通过**protected**成员函数控制对该类的继承者的访问。

```
//: C14:Protected.cpp
// The protected keyword
#include <fstream>
using namespace std;

class Base {
    int i;
protected:
    int read() const { return i; }
    void set(int ii) { i = ii; }
public:
    Base(int ii = 0) : i(ii) {}
    int value(int m) const { return m*i; }
};

class Derived : public Base {
    int j;
public:
    Derived(int jj = 0) : j(jj) {}
    void change(int x) { set(x); }
};

int main() {
    Derived d;
    d.change(10);
} ///:~
```

在本书的后面以及第2卷中，可以看到需要**protected**的例子。

### 14.8.1 protected继承

当继承时，基类默认为**private**，这意味着所有**public**成员函数对于新类的用户是**private**的。通常我们都会按**public**进行继承，从而使得基类的接口也是派生类的接口。然而在继承期间，也可以使用**protected**关键字。

保护继承的派生类意味着对其他类来说是“照此实现”，但它是对于派生类和友元是“**is-a**”。它是不常用的，它的存在只是为了语言的完备性。

## 14.9 运算符的重载与继承

除了赋值运算符以外，其余的运算符可以自动地继承到派生类中。这个可以通过**C12:Byte.h**中的继承加以说明：

```

//: C14:OperatorInheritance.cpp
// Inheriting overloaded operators
#include "../C12/Byte.h"
#include <fstream>
using namespace std;
ofstream out("ByteTest.out");

class Byte2 : public Byte {
public:
    // Constructors don't inherit:
    Byte2(unsigned char bb = 0) : Byte(bb) {}
    // operator= does not inherit, but
    // is synthesized for memberwise assignment.
    // However, only the SameType = SameType
    // operator= is synthesized, so you have to
    // make the others explicitly:
    Byte2& operator=(const Byte& right) {
        Byte::operator=(right);
        return *this;
    }
    Byte2& operator=(int i) {
        Byte::operator=(i);
        return *this;
    }
};

// Similar test function as in C12:ByteTest.cpp:
void k(Byte2& b1, Byte2& b2) {
    b1 = b1 * b2 + b2 % b1;

    #define TRY2(OP) \
        out << "b1 = "; b1.print(out); \
        out << ", b2 = "; b2.print(out); \
        out << "; b1 " #OP " b2 produces "; \
        (b1 OP b2).print(out); \
        out << endl;

    b1 = 9; b2 = 47;
    TRY2(+) TRY2(-) TRY2(*) TRY2(/)
    TRY2(%) TRY2(^) TRY2(&) TRY2(|)
    TRY2(<<) TRY2(>>) TRY2(+=) TRY2(-=)
    TRY2(*=) TRY2(/=) TRY2(%=) TRY2(^=)
    TRY2(&=) TRY2(|=) TRY2(>>=) TRY2(<<=)
    TRY2(=) // Assignment operator

    // Conditionals:
    #define TRYC2(OP) \
        out << "b1 = "; b1.print(out); \
        out << ", b2 = "; b2.print(out); \
        out << "; b1 " #OP " b2 produces "; \
        out << (b1 OP b2); \
        out << endl;

    b1 = 9; b2 = 47;
    TRYC2(<) TRYC2(>) TRYC2(==) TRYC2(!=) TRYC2(<=)
    TRYC2(>=) TRYC2(&&) TRYC2(||)

```



```

    // Chained assignment:
    Byte2 b3 = 92;
    b1 = b2 = b3;
}

int main() {
    out << "member functions:" << endl;
    Byte2 b1(47), b2(9);
    k(b1, b2);
} ///:~

```

除了使用**Byte2**代替了**Byte**以外，该测试代码同C12:ByteTest.cpp中代码是一样的。这种方法通过继承检测了所有运算符是否可以对**Byte2**进行操作。

当检测类**Byte2**时，我们将看到必须显式定义构造函数，同时仅仅生成了可以把**Byte2**赋值于**Byte2**类型的**operator=**，而任何我们需要的赋值运算符将由我们自己生成。

## 14.10 多重继承

既然我们已可以从一个类继承，那么我们就应该能同时从多个类继承。实际上这是可以做到的，但是它像设计部分一样有意义仍是有争议的。不过有一点是肯定的：直到我们已经很好地学会程序设计并完全理解这个语言时，我们才能试着去用它。这时，我们大概会认识到，不管我们如何认为我们必须用多重继承，我们总是能通过单重继承完成。

开始时，多重继承看起来似乎很简单：在继承时，只需在基类列表中增加多个类，用逗号隔开。然而，多重继承引起很多含糊的可能性，这就是为什么要在第2卷中专门有一章讨论这个问题的原因。

## 14.11 渐增式开发

继承和组合的优点之一是它支持渐增式开发 (*incremental development*)，它允许在已存在的代码中引进新代码，而不会给原来的代码带来错误，即使产生了错误，这个错误也只与新代码有关。也就是说通过继承（或通过组合）已存在的功能类并在其基础上增加数据成员和成员函数（并重定义已存在的成员函数）时，已存在类的代码——可能某人仍在使用——并不会被改变，更不会产生错误。如果错误出现，我们就会知道它肯定是在新派生代码中。相对于修改已存在代码体的做法来说，这些新代码很短也很易读。

相当奇怪的是，这些类如何清楚地被隔离。为了重用这些代码，甚至不需要这些成员函数的源代码，只需要表示类的头文件和目标文件或带有已编译成员函数的库文件（对于继承和组合都是这样）。

认识到程序开发就像人的学习过程一样，是一个渐增过程，这是很重要的。我们能做尽可能多的分析，但当开始一个项目时，我们仍不可能知道所有的答案。如果开始把项目作为一个有机的、可进化的生物来“培养”，而不是完全一次性地构造它，像一个玻璃盒子式的摩天大楼，那么我们会获得更大的成功和更直接的反馈<sup>①</sup>。

虽然继承对于实验是有用的技术，但在事情稳定之后，我们需要用新眼光重新审视一下

<sup>①</sup> 为了学习这方面的更多思想，请参见Kent Beck所著的《*Extreme Programming Explained*》(Addison-Wesley 2000)。



我们的类层次，把它看成一个可感知的结构<sup>⑨</sup>。记住，继承首先是表示一种关系，即“新类属于老类的类型（*a type of*）”。我们的程序不应当关心怎样摆布位，而应当关心如何创建和处理各类型的对象，以使用问题空间的术语表示模型。

## 14.12 向上类型转换

在本章的前面，我们已经看到了由**ifstream**派生而来的类的对象如何有**ifstream**对象的所有特性和行为。在**FName2.cpp**中，任何**ifstream**成员函数应当能被**FName2**对象调用。

继承的最重要的方面不是它为新类提供了成员函数，而是它是基类与新类之间的关系，这种关系可被描述为：“新类属于原有类的类型”。

这个描述不仅仅是一种想象的解释继承的方法——它直接由编译器支持。举个例子来说，考虑称为**Instrument**的基类（它表示乐器）和派生类**Wind**（管乐器）。因为继承意味着在基类中的所有函数在派生类中也是可行的，可以发送给基类的消息也可以发送给这个派生类。所以，如果**Instrument**类有**play()**成员函数，那么**Wind**也有。这意味着，可以确切地说，**Wind**对象也就是**Instrument**类型的一个对象。下面的例子表明编译器是如何支持这个概念的。

```
//: C14:Instrument.cpp
// Inheritance & upcasting
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
    void play(note) const {}
};

// Wind objects are Instruments
// because they have the same interface:
class Wind : public Instrument {};

void tune(Instrument& i) {
    // ...
    i.play(middleC);
}

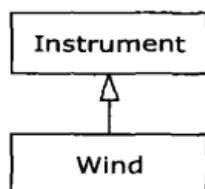
int main() {
    Wind flute;
    tune(flute); // Upcasting
} ///:~
```

在这个例子中，有趣的是**tune()**函数，它接受一个**Instrument**类型的引用。然而，在**main()**中，在**tune()**函数的调用中却被传递了一个**Wind**参数。我们可能会感到奇怪，C++对于类型检查应该是非常严格的，然而接受一个类型的函数为什么会这么容易地接受另一个类型。直到人们认识到**Wind**对象也是一个**Instrument**对象，这里**tune()**函数对于**Instrument**的所有调用对于**Wind**也都是可调用的（这是由继承所保证的）。在**tune()**中，这些代码对**Instrument**和从**Instrument**派生来的任何类型都有效，这种将**Wind**的引用或指针转变成**Instrument**引用或指针的活动被称为向上类型转换（*upcasting*）。

⑨ 参见 Martin Fowler 所著的《*Refactoring: Improving the Design of Existing Code*》(Addison-Wesley, 1999)。

### 14.12.1 为什么要“向上类型转换”

这个术语的引入是有其历史原因的，而且它也与类继承图的传统画法有关：在顶部是根，向下生长（当然我们可以用任何我们认为方便的方法画我们的图）。对于 **Instrument.cpp** 的继承图是



从派生类到基类的类型转换，在继承图上是上升的，所以它一般称为向上类型转换。向上类型转换总是安全的。因为是从更专门的类型到更一般的类型——对于这个类接口可能出现的惟一的事情是它失去成员函数，而不是获得它们。这就是编译器允许向上类型转换而不需要显式地说明或做其他标记的原因。

### 14.12.2 向上类型转换和拷贝构造函数

如果允许编译器为派生类生成拷贝构造函数，它将首先自动地调用基类的拷贝构造函数，然后再是各成员对象的拷贝构造函数（或者在内建类型上执行位拷贝），因此可以得到正确的操作：

```

//: C14:CopyConstructor.cpp
// Correctly creating the copy-constructor
#include <iostream>
using namespace std;

class Parent {
    int i;
public:
    Parent(int ii) : i(ii) {
        cout << "Parent(int ii)\n";
    }
    Parent(const Parent& b) : i(b.i) {
        cout << "Parent(const Parent&)\n";
    }
    Parent() : i(0) { cout << "Parent()\n"; }
    friend ostream&
        operator<<(ostream& os, const Parent& b) {
            return os << "Parent: " << b.i << endl;
        }
};

class Member {
    int i;
public:
    Member(int ii) : i(ii) {
        cout << "Member(int ii)\n";
    }
    Member(const Member& m) : i(m.i) {
        cout << "Member(const Member&)\n";
    }
};
  
```



```

    }
    friend ostream&
        operator<<(ostream& os, const Member& m) {
        return os << "Member: " << m.i << endl;
        }
};

class Child : public Parent {
    int i;
    Member m;
public:
    Child(int ii) : Parent(ii), i(ii), m(ii) {
        cout << "Child(int ii)\n";
    }
    friend ostream&
        operator<<(ostream& os, const Child& c){
        return os << (Parent&)c << c.m
            << "Child: " << c.i << endl;
        }
};

int main() {
    Child c(2);
    cout << "calling copy-constructor: " << endl;
    Child c2 = c; // Calls copy-constructor
    cout << "values in c2:\n" << c2;
} ///:~

```

从对其中**Parent**部分调用**operator<<**的方式可以看出，**Child** 中的**operator<<**很有意思，它通过将**Child**对象类型转换为**Parent&**（但如果是类型转换了一个基类对象，而不是一个引用的话，将得不到所需要的结果）：

```
return os << (Parent&)c << c.m
```

这时编译器把它当做一个**Parent**类型，将调用**operator<<**的**Parent**版本。

我们可以看到**Child**没有显式定义的拷贝构造函数。编译器将通过调用**Parent** 和**Member**的拷贝构造函数来生成它的拷贝构造函数（如果我们没有创建任何构造函数，它是生成的四个函数之一，其他还有默认构造函数、**operator=**和析构函数）。这可从下面的输出中显示出来：

```

Parent(int ii)
Member(int ii)
Child(int ii)
calling copy-constructor:
Parent(const Parent&)
Member(const Member&)
values in c2:
Parent: 2
Member: 2
Child: 2

```

然而，如果试着为**Child**写自己的拷贝构造函数，并且出现错误：

```
Child(const Child& c) : i(c.i), m(c.m) {}
```

这时将会为**Child**中的基类部分调用默认的构造函数，这是在没有其他的构造函数可供选择调用的情况下，编译器回溯搜索的结果（记住某些构造函数总是必须为每个对象所调用，

而不管它是否是一个其他类的子对象)。这样输出将会是：

```
Parent(int ii)
Member(int ii)
Child(int ii)
calling copy-constructor:
Parent()
Member(const Member&)
values in c2:
Parent: 0
Member: 2
Child: 2
```

这可能并不是我们所希望的，因为通常我们会希望基类部分从已存在对象拷贝至一个新的对象，以作为拷贝构造函数的一部分。

为了解决这个问题，必须记住无论何时我们在创建了自己的拷贝构造函数时，都要正确地调用基类拷贝构造函数（正如编译器所作的）。这乍一看可能有点奇怪，但它是向上类型转换的另一种情况：

```
Child(const Child& c)
: Parent(c), i(c.i), m(c.m) {
    cout << "Child(Child&)\n";
}
```

奇怪的部分在于调用**Parent**的拷贝构造函数的地方：**Parent(c)**。传送一个**Child**对象给**Parent**构造函数意味着什么？因为**Child**是由**Parent**继承而来，所以**Child**的引用也就相当于**Parent**的引用。基类拷贝构造函数的调用将一个**Child**的引用向上类型转换为一个**Parent**的引用，并且使用它来执行拷贝构造函数。当我们创建自己的拷贝构造函数时，也总会做同样的事情。

### 14.12.3 组合与继承（再论）

确定应当用组合还是用继承，最清楚的方法之一是询问是否需要从新类向上类型转换。在本章的前面，**Stack**类通过继承被专门化，然而，**StringStack**对象仅作为**string**容器，不需向上类型转换，所以更合适的方法可能是组合：

```
//: C14:InheritStack2.cpp
// Composition vs. inheritance
#include "../C09/Stack4.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class StringStack {
    Stack stack; // Embed instead of inherit
public:
    void push(string* str) {
        stack.push(str);
    }
    string* peek() const {
        return (string*)stack.peek();
    }
}
```



```

    }
    string* pop() {
        return (string*)stack.pop();
    }
};

int main() {
    ifstream in("InheritStack2.cpp");
    assure(in, "InheritStack2.cpp");
    string line;
    StringStack textlines;
    while(getline(in, line))
        textlines.push(new string(line));
    string* s;
    while((s = textlines.pop()) != 0) // No cast!
        cout << *s << endl;
} ///:~

```

这个文件与**InheritStack.cpp**是一样的，只不过**Stack**对象被嵌入在**StringStack**内，并且成员函数是由被嵌入对象调用的。这里没有时间和空间的开销，因为其子类占用相同量的空间，而且所有另外的类型检查是发生在编译时。

虽然这可能会变得更加复杂，但我们可以用**private**继承以表示“照此实现”，这也将很好地解决了这个问题。然而，一个重要的方面是确保多重继承。在这种情况下，如果发现一个程序中可以使用组合来代替继承，我们便可以消除对多重继承的需要。

#### 14.12.4 指针和引用的向上类型转换

在**Instrument.cpp**中，向上类型转换发生在函数调用期间——在函数外的**Wind**对象被引用并且变成一个在这个函数内的**Instrument**的引用。向上类型转换还能出现在对指针或引用简单赋值期间：

```

Wind w;
Instrument* ip = &w; // Upcast
Instrument& ir = w; // Upcast

```

与函数调用一样，这两个例子都不要求显式地类型转换。

#### 14.12.5 危机

当然，任何向上类型转换都会损失对象的类型信息，如果如下编写：

```

Wind w;
Instrument* ip = &w;

```

则编译器只能把**ip**作为一个**Instrument**指针处理。这就是，它不能知道**ip**实际上可能是指向**Wind**的对象。所以，当调用**play()**成员函数时，使用

```
ip->play(middleC);
```

编译器只能知道它正在对于一个**Instrument**指针调用**play()**，并调用**Instrument::play()**的基本版本，而不是它应该做的调用**wind::play()**。这样将会得到不正确的结果。

这是一个重要的问题，将在第15章通过介绍面向对象编程的第三块基石：多态性（在C++中用**virtual**函数实现）来解决。

### 14.13 小结

继承和组合都允许由已存在的类型创建新类型，两者都是在新类型中嵌入已存在的类型的子对象。然而，如果想重用已存在类型作为新类型的内部实现的话，我们最好用组合；如果想使新的类型和基类的类型相同（类型一样可确保接口一样），则应使用继承。如果派生类有基类的接口，它就能向上类型转换到这个基类，这一点对第15章中介绍的多态性很重要。

虽然通过组合和继承进行代码重用对于快速项目开发有帮助，但通常我们会希望在允许其他程序员依据它开发之前重新设计类层次。我们的类层次必须有这样的特性：它的每个类有专门的用途，它不能太大（包含太多不利于重用的功能），也不能太小（太小如不对它本身增加功能就不能使用）。

### 14.14 练习

部分练习题的答案可以在本书的电子文档“*Annotated Solution Guide for Thinking in C++*”中找到，只需支付很少的费用就可以在<http://www.BruceEckel.com>得到这个电子文档。

- 14-1 修改**Car.cpp**，使它也继承**Vehicle**类，在**Vehicle**中放置合适的成员函数（也就是说，补充一些成员函数）。对**Vehicle**增加一个非默认的构造函数，在**Car**的构造函数内部必须调用它。
- 14-2 创建两个类，**A**和**B**，带有默认的构造函数。从**A**继承出一个新类，称为**C**，并且在**C**中创建**B**的一个成员对象，而不对**C**创建构造函数。创建类**C**的一个对象，观察结果。
- 14-3 创建一个三层的类结构，带有默认的构造函数和析构函数，它们都对**cout**做了声明。对于最底层的派生类对象，验证所有三个构造函数和析构函数都自动被调用。解释这里调用的顺序。
- 14-4 修改**Combined.cpp**，再多继承一层并且增加一个新的成员对象。添加代码来显示何时调用构造函数和析构函数。
- 14-5 在**Combined.cpp**中，创建从类**B**继承来的类**D**，它含有一个类**C**的成员对象。添加代码来显示何时调用构造函数和析构函数。
- 14-6 修改**Order.cpp**，再继承出一层，即**Derived3**，它含有类**Member4**和类**Member5**的成员对象。跟踪程序的输出。
- 14-7 在**NameHiding.cpp**中验证，**Derived2**、**Derived3** 和 **Derived4** 中**f()**的基类版本都是不可用的。
- 14-8 修改**NameHiding.cpp**，在**Base**中增加三个名为**h()**的重载函数。然后显示在派生类中重新定义其中的一个函数，则会隐藏其余的函数。
- 14-9 从**vector<void\*>**中继承出类**StringVector**，重新定义**push\_back()**和 **operator[]**成员函数以接收和生成**string\***。如果试着**push\_back()**一个**void\***，会发生什么情况？
- 14-10 创建一个包含**long**型成员的类，对构造函数使用psuedo构造函数调用语法来初始化这个**long**成员。
- 14-11 创建类**Asteroid**，使用继承，具体在第13章（**PStash.h** & **PStash.cpp**）中的**PStash**类，使得它接受和返回**Asteroid**指针。修改**PStashtest.cpp**并测试该类。改变这个类使得**PStash**是一个成员对象。

- 14-12 使用**vector**来代替**PStash**，重复练习11。
- 14-13 在**SynthesizedFunctions.cpp**中，修改**Chess**，使它有一个默认构造函数、拷贝构造函数和赋值运算符。显示我们进行了正确的修改。
- 14-14 创建两个不含有默认构造函数的类**Traveler**和**Pager**，但具有一个参数为**string**的构造函数，该构造函数只是简单地把**string**参数拷贝至一个内部变量中。对于每个类，创建正确的拷贝构造函数和赋值运算符。现在从**Traveler**中继承出类**BusinessTraveler**，并使其包含一个类**Pager**的对象。创建正确的默认构造函数、参数为**string**的构造函数、拷贝构造函数和赋值运算符。
- 14-15 创建含有两个**static**成员函数的类。继承这个类，并且重新定义其中一个成员函数，显示出另一个函数在派生类中被隐藏。
- 14-16 找出**ifstream**更多的成员函数。在**FName2.cpp**中，尝试将它们输出在**file**对象上。
- 14-17 使用**private**和**protected**继承方式从基类中创建两个新类。然后试着把派生类的对象向上类型转换为基类对象。解释所发生的事情。
- 14-18 在**Protected.cpp**中，在**Derived**里增加一个成员函数，它用来调用**Base**类中的**protected**成员函数**read()**。
- 14-19 修改**Protected.cpp**使得**Derived**是按**protected**方式继承来的。看看一个**Derived**对象是否可以调用**value()**。
- 14-20 创建一个含有**fly()**方法的类**SpaceShip**。从**SpaceShip**中继承出**Shuttle**，并且增加一个方法**land()**。创建一个新的类**Shuttle**，通过一个**SpaceShip**对象的指针或引用向上类型转换，并且试着调用**land()**方法。解释操作的结果。
- 14-21 修改**Instrument.cpp**，对**Instrument**增加一个**prepare()**方法。在**tune()**中调用**prepare()**。
- 14-22 修改**Instrument.cpp**，使**play()**向**cout**打印出消息，并且**Wind**重新定义了**play()**，使之向**cout**打印出不同的消息。运行这个程序并解释为什么我们不想有这样的结果。然后在**Instrument**中**play()**的声明前加上**virtual**关键字（我们将在第15章中学习），观察结果有什么不同。
- 14-23 在**CopyConstructor.cpp**中，由**Child**继承出一个新类，使其具有一个**Member m**。并且创建适当的构造函数、拷贝构造函数、**operator=**和用于**ostreams**的**operator<<**。在**main()**中测试这个类。
- 14-24 修改例子**CopyConstructor.cpp**，对**Child**使用我们自己的拷贝构造函数，它不调用基类拷贝构造函数，看看有什么情况发生。在**Child**的拷贝构造函数中的初始化列表里，通过进行适当的显式地对基类拷贝构造函数的调用来分析和解决这个问题。
- 14-25 使用**vector<string>**代替**Stack**，来对**InheritStack2.cpp**进行修改。
- 14-26 创建一个类**Rock**，含有默认构造函数、拷贝构造函数、赋值运算符和析构函数，它们都向**cout**声明它们已经被调用。在**main()**中，创建一个**vector<Rock>**（即通过传值方式得到**Rock**对象）并且添加一些**Rock**对象。运行这个程序并解释我们得到的输出。注意**vector**里所有**Rock**的析构函数是否都被调用了。然后用**vector<Rock\*>**来重复上面的操作。可以创建**vector<Rock&>**吗？
- 14-27 本练习创建一个名为代理（*proxy*）的设计模式。首先建立一个基类**Subject**，使其包含

三个函数 **f()**、**g()** 和 **h()**。现在从中继承出类 **Proxy** 和另外两个类 **Implementation1** 和 **Implementation2**。**Proxy** 中应包含指向 **Subject** 的指针，于是它的所有成员函数可以通过该 **Subject** 指针指回，调用 **Subject** 的函数。**Proxy** 的构造函数的参数是指向 **Subject** 的指针，它被包含在 **Proxy** 内（通常这由构造函数完成）。在 **main()** 中，使用两种不同的实现方式创建两个不同的 **Proxy** 对象。然后修改 **Proxy** 使得我们可以动态地改变实现方式。

- 14-28 修改第13章的 **ArrayOperatorNew.cpp** 以显示，如果我们继承 **Widget**，则仍将可以正确地执行分配。解释为什么不能正确地执行第13章中 **Framis.cpp** 的继承。
- 14-29 修改第13章的 **Framis.cpp**，对 **Framis** 进行继承，并且为我们的派生类创建新版本的 **new** 和 **delete**。表明可以正确地运行它们。





## 多态性和虚函数

多态性（在C++中通过虚函数来实现）是面向对象程序设计语言中数据抽象和继承之外的第三个基本特征。

多态性（polymorphism）提供了接口与具体实现之间的另一层隔离，从而将“*what*”与“*how*”分离开来。多态性改善了代码的组织性和可读性，同时也使创建的程序具有可扩展性，程序不仅在项目的最初创建期可以“扩展”，而且当在项目需要有新的功能时也能“扩展”。

封装（encapsulation）通过组合特性和行为来生成新的数据类型。访问控制通过使细节数据设为**private**，将接口从具体实现中分离开来。这类机制对于具有过程化程序设计背景的人来说是非常有意义的。而虚函数则根据类型来处理解耦。在第14章中，我们已经看到，如何继承通过把对象作为它自己的类型或它的基类型来处理。这种能力非常关键，因为它允许很多类型（从同一个基类型派生）被看做是一个类型，一段代码可以同样地工作在所有这些不同类型上。虚函数允许一个类型表达自己与另一个相似类型之间的区别，只要这两个类型都是从同一个基类型派生的。这种区别是通过从基类调用的那些函数行为的不同来表达的。

在本章中，我们将从基本知识开始学习虚函数，为了简单起见，本章所用的例子经过简化，只保留了程序的“虚”性质。

### 15.1 C++程序员的演变

C程序员可以用三步演变为C++程序员。第一步：简单地把C++作为一个“更好的C”，因为C++要求在使用任何函数之前必须声明它，并且对于如何使用变量有更苛刻的要求。简单地用C++编译器编译C程序常常会发现错误。

第二步：进入“基于对象”的C++。这意味着，很容易看到将数据结构和在它上面活动的函数捆绑在一起的代码组织好处，还可以看到构造函数和析构函数的价值，也许还会看到一些简单的继承。大多数用过C工作的程序员很快就看到这个步骤是有用的，因为无论何时，当他们创建库时，这个步骤都是他们努力要去做。然而在C++中，将由编译器来帮助我们完成这个步骤。

在基于对象的层面上，我们容易产生错觉，因为我们可以很快成功，并且无需花费太多精力就能得到很多好处。感觉就好像我们正在创建数据类型——制造类和对象，向这些对象发送消息，一切恰到好处并且干净利落。

但是，不要犯傻。如果我们停留在这里，我们就会错失这个语言最重要的部分。这个最重要的部分才是通向真正的面向对象程序设计的飞跃。要做到这一点，只有靠虚函数。

虚函数增强了类型概念，而不是只在结构内部隐蔽地封装代码，所以毫无疑问，对于新的C++程序员来说，这些概念是最困难的。然而，它们也是理解面向对象程序设计的转折点。如果不用虚函数，就等于还不懂得面向对象程序设计（OOP）。

因为虚函数是与类概念紧密联系的，而类是面向对象程序设计的核心，所以在传统的过程型语言中没有类似于虚函数的东西。作为一个过程型程序员，没有什么事物可以帮助他思考虚函数，这与接触该语言的其他大多数功能还有所参照的情况大为不同。过程型语言中的特征可以在算法层上来理解，而虚函数只能从设计的观点来理解。

## 15.2 向上类型转换

在第14章中，我们已经看到对象如何能作为它自己的类或作为它的基类的对象来使用。另外，还能通过基类的地址操作它。取一个对象的地址（指针或引用），并将其作为基类的地址来处理，这被称为向上类型转换（*upcasting*），因为继承树的绘制方式是以基类为顶点的。

我们还看到出现一个问题，它体现在如下的代码段中：

```
//: C15:Instrument2.cpp
// Inheritance & upcasting
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Eflat }; // Etc.

class Instrument {
public:
    void play(note) const {
        cout << "Instrument::play" << endl;
    }
};

// Wind objects are Instruments
// because they have the same interface:
class Wind : public Instrument {
public:
    // Redefine interface function:
    void play(note) const {
        cout << "Wind::play" << endl;
    }
};

void tune(Instrument& i) {
    // ...
    i.play(middleC);
}

int main() {
    Wind flute;
    tune(flute); // Upcasting
} ///:~
```

函数 `tune()`（通过引用）接受一个 `Instrument`，但也不拒绝任何从 `Instrument` 派生的类。在 `main()` 中，可以看到，无须类型转换，就能将 `Wind` 对象传给 `tune()`。这是可接受的；在 `Instrument` 中的接口必然存在于 `Wind` 中，因为 `Wind` 是从 `Instrument` 中按公有方式继承而来的。`Wind` 到 `Instrument` 的向上类型转换会使 `Wind` 的接口“变窄”，但不会窄过 `Instrument` 的整个接口。

处理指针时采用相同的参数；惟一的不同是用户必须显式地取对象的地址传给函数。

## 15.3 问题

运行程序**Instrument2.cpp**可以看到这个程序中的问题。调用输出的是**Instrument::play**。显然，这不是所希望的输出，因为我们知道这个对象实际上是**Wind**而不是一个**Instrument**。应当调用的是**Wind::play**。为此，由**Instrument**派生的任何对象不论它处于什么位置都应当使用它的**play()**版本。

然而，当对函数用C方法时，**Instrument2.cpp**的行为并不使人惊奇。为了理解这个问题，需要知道捆绑 (*binding*) 的概念。

### 15.3.1 函数调用捆绑

把函数体与函数调用相联系称为捆绑 (*binding*)。当捆绑在程序运行之前 (由编译器和连接器) 完成时，这称为早捆绑 (*early binding*)。我们可能没有听过这个术语，因为在过程型语言中不会有这样的选择：C编译只有一种函数调用方式，就是早捆绑。

上面程序中的问题是早捆绑引起的，因为编译器在只有**Instrument**地址时它并不知道要调用的正确函数。

解决方法被称为晚捆绑 (*late binding*)，这意味着捆绑根据对象的类型，发生在运行时。晚捆绑又称为动态捆绑 (*dynamic binding*) 或运行时捆绑 (*runtime binding*)。当一个语言实现晚捆绑时，必须有某种机制来确定运行时对象的类型并调用合适的成员函数。对于一种编译语言，编译器并不知道实际的对象类型，但它插入能找到和调用正确函数体的代码。晚捆绑机制因语言而异，但可以想象，某些种类的类型信息必须装在对象自身中。稍后将会看到它是如何工作的。

## 15.4 虚函数

对于特定的函数，为了引起晚捆绑，C++要求在基类中声明这个函数时使用**virtual**关键字。晚捆绑只对**virtual**函数起作用，而且只在使用含有**virtual**函数的基类的地址时发生，尽管它们也可以在更早的基类中定义。

为了创建一个像**virtual**这样的成员函数，可以简单地在这个函数声明的前面加上关键字**virtual**。仅仅在声明的时候需要使用关键字**virtual**，定义时并不需要。如果一个函数在基类中被声明为**virtual**，那么在所有的派生类中它都是**virtual**的。在派生类中**virtual**函数的重定义通常称为重写 (*overriding*)。

注意，仅需要在基类中声明一个函数为**virtual**。调用所有匹配基类声明行为的派生类函数都将使用虚机制。虽然可以在派生类声明前使用关键字**virtual** (这也是无害的)，但这样会使程序段显得冗余和混乱。

为了从**Instrument2.cpp**中得到所希望的结果，只需简单地在基类中的**play()**之前增加**virtual**关键字：

```
//: C15:Instrument3.cpp
// Late binding with the virtual keyword
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat }; // Etc.
```

```

class Instrument {
public:
    virtual void play(note) const {
        cout << "Instrument::play" << endl;
    }
};

// Wind objects are Instruments
// because they have the same interface:
class Wind : public Instrument {
public:
    // Override interface function:
    void play(note) const {
        cout << "Wind::play" << endl;
    }
};

void tune(Instrument& i) {
    // ...
    i.play(middleC);
}

int main() {
    Wind flute;
    tune(flute); // Upcasting
} ///:~

```

这个文件除了增加了**virtual**关键字之外，一切与**Instrument2.cpp**相同，但结果明显不一样。现在输出调用的是**Wind::play**。

#### 15.4.1 扩展性

通过将**play()**在基类中定义为**virtual**，不用改变**tune()**函数就可以在系统中随意增加新函数。在一个设计风格良好的 OOP 程序中，大多数甚至所有的函数都沿用**tune()**模型，只与基类接口通信。这样的程序是可扩展的 (*extensible*)，因为可以通过从公共基类继承新数据类型而增加新功能。操作基类接口的函数完全不需要改变就可以适合于这些新类。

这里有一个**instrument**例子，它有更多的虚函数和一些新类，它们都能与老的版本一起正确工作，而不用改变**tune()**函数：

```

//: C15:Instrument4.cpp
// Extensibility in OOP
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
    virtual void play(note) const {
        cout << "Instrument::play" << endl;
    }
    virtual char* what() const {
        return "Instrument";
    }
}

```



```

    // Assume this will modify the object:
    virtual void adjust(int) {}
};

class Wind : public Instrument {
public:
    void play(note) const {
        cout << "Wind::play" << endl;
    }
    char* what() const { return "Wind"; }
    void adjust(int) {}
};

class Percussion : public Instrument {
public:
    void play(note) const {
        cout << "Percussion::play" << endl;
    }
    char* what() const { return "Percussion"; }
    void adjust(int) {}
};

class Stringed : public Instrument {
public:
    void play(note) const {
        cout << "Stringed::play" << endl;
    }
    char* what() const { return "Stringed"; }
    void adjust(int) {}
};

class Brass : public Wind {
public:
    void play(note) const {
        cout << "Brass::play" << endl;
    }
    char* what() const { return "Brass"; }
};

class Woodwind : public Wind {
public:
    void play(note) const {
        cout << "Woodwind::play" << endl;
    }
    char* what() const { return "Woodwind"; }
};

// Identical function from before:
void tune(Instrument& i) {
    // ...
    i.play(middleC);
}

// New function:
void f(Instrument& i) { i.adjust(1); }

```



```
// Upcasting during array initialization:
Instrument* A[] = {
    new Wind,
    new Percussion,
    new Stringed,
    new Brass,
};

int main() {
    Wind flute;
    Percussion drum;
    Stringed violin;
    Brass flugelhorn;
    Woodwind recorder;
    tune(flute);
    tune(drum);
    tune(violin);
    tune(flugelhorn);
    tune(recorder);
    f(flugelhorn);
} ///:~
```

可以看到，这个例子已在**Wind**之下增加了另外的继承层，但不管这里有多少层，**virtual**机制仍会正确工作。针对**Brass**和**Woodwind**，**adjust()**函数没有重写（重新定义）。当出现这种情况时，将会自动地调用继承层次中“最近”的定义——编译器保证对于虚函数总是有某种定义，所以决不会出现最终调用不与函数体捆绑的情况（这种情况将导致灾难）。

数组**A[]**存放指向基类**Instrument**的指针，所以在数组初始化过程中发生向上类型转换。这个数组和函数**f()**将在稍后的讨论中用到。

在对**tune()**的调用中，向上类型转换在对象的每一个不同的类型上完成。总能得到期望的结果。这可以被描述为“发送一条消息给一个对象，让这个对象考虑用它来做什么”。**virtual**函数使我们在分析项目时可以初步确定：基类应当出现在哪里？应当如何扩展这个程序？在程序最初创建时，即便我们没有发现合适的基类接口和虚函数，但在稍后或者更晚，当决定扩展或维护这个程序时，也常常会发现它们。这不是分析或设计错误，它只意味着一开始我们还没有所有的信息。由于C++中严格的模块化，因此这并不是大问题。因为当我们对系统的一部分进行修改时，往往不会像C那样波及系统的其他部分。

## 15.5 C++如何实现晚捆绑

晚捆绑如何发生？所有的工作都由编译器在幕后完成。当告诉编译器要晚捆绑时（通过创建虚函数来告诉），编译器安装必要的晚捆绑机制。因为程序员常常从理解C++虚函数机制中受益，所以这一节将详细阐述编译器实现这一机制的方法。

关键字**virtual**告诉编译器它不应当执行早捆绑，相反，它应当自动安装对于实现晚捆绑必需的所有机制。这意味着，如果对**Brass**对象通过基类**Instrument**地址调用**play()**，将得到恰当的函数。

为了达到这个目的，典型的编译器<sup>①</sup>对每个包含虚函数的类创建一个表（称为**VTABLE**）。

① 编译器可以按它们希望的任何方式执行虚操作，但是这里所讨论的方法是一种相当通用的方法。

在VTABLE中，编译器放置特定类的虚函数的地址。在每个带有虚函数的类中，编译器秘密地放置一个指针，称为`vpointer`（缩写为VPTR），指向这个对象的VTABLE。当通过基类指针做虚函数调用时（也就是做多态调用时），编译器静态地插入能取得这个VPTR并在VTABLE表中查找函数地址的代码，这样就能调用正确的函数并引起晚捆绑的发生。

为每个类设置VTABLE、初始化VPTR、为虚函数调用插入代码，所有这些都是自动发生的，所以不必担心。利用虚函数，即使在编译器还不知道这个对象的特定类型的情况下，也能调用这个对象中正确的函数。

下面几节将进行更详细的阐述。

### 15.5.1 存放类型信息

可以看到，在任何类中不存在显式的类型信息。而先前的例子和简单的逻辑告诉我们，必须有一些类型信息放在对象中；否则，类型将不能在运行时被建立。确实是这样的，但类型信息被隐藏了。为了看到这些信息，这里举一个例子，以便检查使用虚函数的类的长度，并与没有虚函数的类进行比较。

```
//: C15:Sizes.cpp
// Object sizes with/without virtual functions
#include <iostream>
using namespace std;

class NoVirtual {
    int a;
public:
    void x() const {}
    int i() const { return 1; }
};

class OneVirtual {
    int a;
public:
    virtual void x() const {}
    int i() const { return 1; }
};

class TwoVirtuals {
    int a;
public:
    virtual void x() const {}
    virtual int i() const { return 1; }
};

int main() {
    cout << "int: " << sizeof(int) << endl;
    cout << "NoVirtual: "
        << sizeof(NoVirtual) << endl;
    cout << "void* : " << sizeof(void*) << endl;
    cout << "OneVirtual: "
        << sizeof(OneVirtual) << endl;
    cout << "TwoVirtuals: "
        << sizeof(TwoVirtuals) << endl;
} ///:~
```

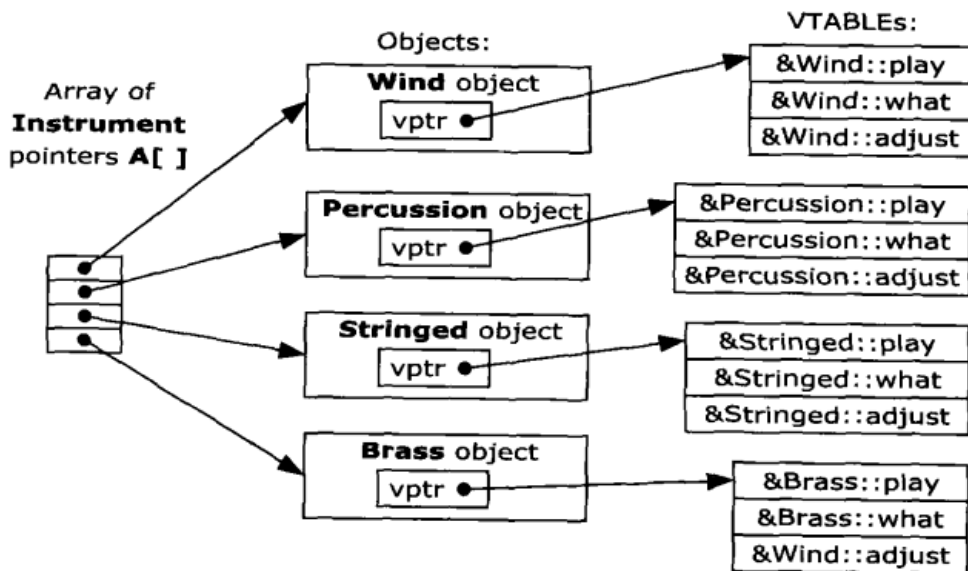


不带虚函数，对象的长度恰好就是所期望的长度：单个<sup>①</sup>`int`的长度。而带有单个虚函数的`OneVirtual`，对象的长度是`NoVirtual`的长度加上一个`void`指针的长度。它反映出，如果有一个或多个虚函数，编译器都只在这个结构中插入一个单个指针（VPTR）。因此`OneVirtual`和`TwoVirtuals`的长度没有区别。这是因为VPTR指向一个存放函数地址的表。我们只需要一个表，因为所有虚函数地址都包含在这个单个表中。

这个例子至少要求一个数据成员。如果没有数据成员，C++编译器会强制这个对象是非零长度，因为每个对象必须有一个互相区别的地址。如果我们想象在一个零长度对象的数组中索引寻址，就能理解这一点。把一个“哑”成员插入到对象中，否则这个对象就会是零长度。当类型信息由于存在这个关键字`virtual`而被插入时，这个“哑”成员的位置就被占用。在上例中，用注释符号将`int a`这一行去掉，就会看到这种情况。

### 15.5.2 虚函数功能图示

下面是`Instrument4.cpp`中的指针数组`A[ ]`的图，它可以帮助我们准确地理解当使用虚函数时编译器进行的内部活动。



这个`Instrument`指针数组没有特殊类型信息，它的每一个元素都指向一个类型为`Instrument`的对象。`Wind`、`Percussion`、`Stringed`和`Brass`都可以归入这个类别之中，因为它们都是从`Instrument`派生来的（并因而与`Instrument`有相同的接口和可以响应相同的消息），所以它们的地址自然被放进这个数组。然而，编译器并不知道它们是比`Instrument`对象具有更多内容东西，所以，就将它们留给其自己的设备处理，而通常调用所有函数的基类版本。但在这里，所有这些函数都被用`virtual`声明，所以出现了不同的情况。

每当创建一个包含有虚函数的类或从包含有虚函数的类派生一个类时，编译器就为这个类创建一个惟一的VTABLE，如这个图的右面所示。在这个表中，编译器放置了在这个类中或在它的基类中所有已声明为`virtual`的函数的地址。如果在这个派生类中没有对在基类中声明为`virtual`的函数进行重新定义，编译器就使用基类的这个虚函数地址。（在`Brass`的VTABLE中，

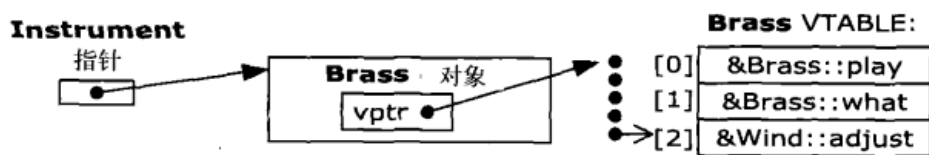
<sup>①</sup> 这里某些编译器可能含有长度功能，但是并不多见。



**adjust**的入口就是这种情况。)然后编译器在这个类中放置VPTR (可在**Sizes.ccp**中发现)。当使用简单继承时,对于每个对象只有一个VPTR。VPTR必须被初始化为指向相应的VTABLE的起始地址。(这在构造函数中发生,在稍后会看得更清楚。)

一旦VPTR被初始化为指向相应的VTABLE,对象就“知道”它自己是什么类型。但只有当虚函数被调用时这种自我认知才有用。

当通过基类地址调用一个虚函数时(此时编译器没有能完成早捆绑所需的所有信息),要特殊处理。它不是实现典型的函数调用,那样只是简单地用汇编语言**CALL**特定的地址,而是编译器为完成这个函数调用而产生不同的代码。下面看到的是通过**Instrument**指针对于**Brass**调用**adjust()**(**Instrument**引用产生同样的结果)。



编译器从这个**Instrument**指针开始,这个指针指向这个对象的起始地址。对于所有的**Instrument**对象或由**Instrument**派生的对象,它们的VPTR都在对象的不同位置(常常在对象的开头),所以编译器能够取出这个对象的VPTR。VPTR指向VTABLE的起始地址。所有的VTABLE都具有相同的顺序,不管何种类型的对象。**play()**是第一个,**what()**是第二个,**adjust()**是第三个。所以无论什么特殊的对象类型,编译器都知道**adjust()**函数必在VPTR+2处。这样,不是“以**Instrument::adjust**地址调用这个函数”(这是早捆绑,是错误动作),而是产生代码,即实际上“在VPTR+2处调用这个函数”。因为获取VPTR和确定实际函数地址发生在运行时,所以这样就得到了所希望的晚捆绑。我们向这个对象发送消息,随后这个对象能断定它应当做什么。

### 15.5.3 撩开面纱

如果能看到由虚函数调用而产生的汇编语言代码,这将是很有帮助的,这样我们可以看到晚捆绑实际上是如何发生的。下面是在函数**f(Instrument&i)**内部调用:

```
i.adjust(1);
```

某个编译器所产生的输出:

```
push 1
push si
mov bx, word ptr [si]
call word ptr [bx+4]
add sp, 4
```

C++函数调用的参数与C函数调用一样,是从右向左进栈的(这个顺序是为了支持C的变量参数表),所以参数**1**首先压栈。对于这个函数,寄存器**si**(Intel x86处理器的一部分)存放**i**的地址。因为它是被选中的对象的首地址,它也被压进栈。记住,这个首地址对应于**this**的值,正因为调用每个成员函数时**this**都必须作为参数压进栈,所以成员函数知道它工作在哪个特殊对象上。这样,我们总能看到,在成员函数调用之前压栈的次数等于参数个数加1(除了**static**成员函数,它没有**this**)。

现在,必须实现实际的虚函数调用。首先,必须产生VPTR,使得能找到VTABLE。对于

这个编译器，VPTR在对象的开头，所以**this**的内容对应于VPTR。下面这一行：

```
mov bx, word ptr [si]
```

取出**si**（即**this**）所指的字，它就是VPTR。将这个VPTR放入寄存器**bx**中。

放在**bx**中的这个VPTR指向这个VTABLE的首地址，但被调用的函数不是在VTABLE中第0个位置，而是在第2个位置（因为它是这个表中的第3个函数）。对于这种内存模式，每个函数指针是两个字节长，所以编译器对VPTR加4，计算相应的函数地址所在的地方。注意，这是编译时建立的常值，所以惟一要做的事情就是保证在第2个位置上的指针恰好指向**adjust()**。幸好编译器仔细处理，并保证在VTABLE中的所有函数指针都以相同的次序出现，而不论我们在派生类中是以什么次序覆盖它们。

一旦在VTABLE中相应函数指针的地址被计算出来，就调用这个函数。所以取出这个地址并马上在这个句子中调用：

```
call word ptr [bx+4]
```

最后，栈指针移回去，以清除在调用之前压入栈的参数。在C和C++汇编代码中，将经常看到调用者清除这些参数，但这可能依据处理器和编译器的实现而有所不同。

#### 15.5.4 安装vpointer

因为VPTR决定了对象的虚函数的行为，所以我们可以看到VPTR总是指向相应的VTABLE是多么重要。在VPTR适当初始化之前绝对不能调用虚函数。当然，可以保证初始化的地点是在构造函数中，但是在**Instrument**例子中没有一个是具有构造函数的。

这样，默认构造函数的创建是很关键的。在**Instrument**例子中，编译器创建了一个默认构造函数，它只做初始化VPTR的工作。在使用任何**Instrument**对象之前，对于**Instrument**对象自动调用这个构造函数。所以，可以安全地调用虚函数。

在下一节中我们将讨论在构造函数内部自动初始化VPTR的含义。

#### 15.5.5 对象是不同的

认识到向上类型转换仅处理地址，这是重要的。如果编译器有一个它知道确切类型的对象，那么（在C++中）对任何函数的调用将不再使用晚捆绑，或至少编译器不必使用晚捆绑。因为编译器知道对象的确切类型，为了提高效率，当调用这些对象的虚函数时，很多编译器使用早捆绑。下面是一个例子：

```
//: C15:Early.cpp
// Early binding & virtual functions
#include <iostream>
#include <string>
using namespace std;

class Pet {
public:
    virtual string speak() const { return ""; }
};

class Dog : public Pet {
public:
```



```

    string speak() const { return "Bark!"; }
};

int main() {
    Dog ralph;
    Pet* p1 = &ralph;
    Pet& p2 = ralph;
    Pet p3;
    // Late binding for both:
    cout << "p1->speak() = " << p1->speak() << endl;
    cout << "p2.speak() = " << p2.speak() << endl;
    // Early binding (probably):
    cout << "p3.speak() = " << p3.speak() << endl;
} ///:~

```

在`p1->speak()`和`p2.speak()`中，使用地址，就意味着信息不完全：`p1`和`p2`可能表示`Pet`的地址，也可能表示其派生对象的地址，所以必须使用虚函数。而当调用`p3.speak()`时不存在含糊，编译器知道确切的类型且知道它是一个对象，所以它不可能由`Pet`派生的对象，而确切的只是一个`Pet`。这样，可以使用早捆绑。但是，如果不希望编译器的工作如此复杂，仍然可以使用晚捆绑，并且会产生相同的行为。

## 15.6 为什么需要虚函数

在这个问题上，我们可能会问：“如果这个技术如此重要，并且使得任何时候都能调用‘正确’的函数，那么为什么它是可选的呢？为什么我甚至还需要知道它呢？”

问得好。回答关系到C++的基本哲学：“因为它不是相当高效的”。从前面的汇编语言输出可以看出，它并不是对于绝对地址的一个简单的CALL，而是为设置虚函数调用需要两条以上的复杂的汇编指令。这既需要代码空间，又需要执行时间。

一些面向对象的语言已经接受了这种途径，即晚捆绑对于面向对象程序设计是性质所固有的，所以应当总是出现，它不应当是可选的，而且用户并不一定需要知道它。这是在创造语言的设计时决定的，而这种特殊的方法对于许多语言是合适的。<sup>①</sup>而C++来自于C，在C中，效率是重要的。创造C完全是为了代替汇编语言以实现操作系统（从而改写操作系统——UNIX——使得比它的先驱更轻便）。发明C++的主要原因之一是让C程序员的工作具有更高效率。<sup>②</sup>当C程序员遇到C++时要问的第一个问题是“我将得到什么样的规模和速度效果”？如果回答是“除了函数调用时需要有一点额外的开销外，一切皆好”，那么许多人就会仍使用C，而不会改变到C++。另外，内联函数是不可能的，因为虚函数必须有地址放在VTABLE中。所以虚函数是可选的，而且该语言的默认是非虚拟的，这是最快的配置。Stroustrup声明他的方针是，“如果我们不用它，我们就不会为它花费额外的开销。”

因此，**virtual**关键字可以改变程序的效率。然而，当设计类时，我们不当为效率问题担心。如果想使用多态，就在每处使用虚函数。当试图加速代码时，只需寻找可以不使用虚函数的函数（而且通常可能在其他方面获得更大收益——好的编程者会在查找瓶颈方面，而不是在猜测方面投入更多的工作）。

① 例如，Smalltalk、Java及Python语言都成功地使用了这种方法。

② 在C++的发源地——贝尔实验室中，汇集着大量的C程序员。尽力使这些C程序员的工作更加有效率，即使是改善一点点，也会为公司节省数百万美元的开销。

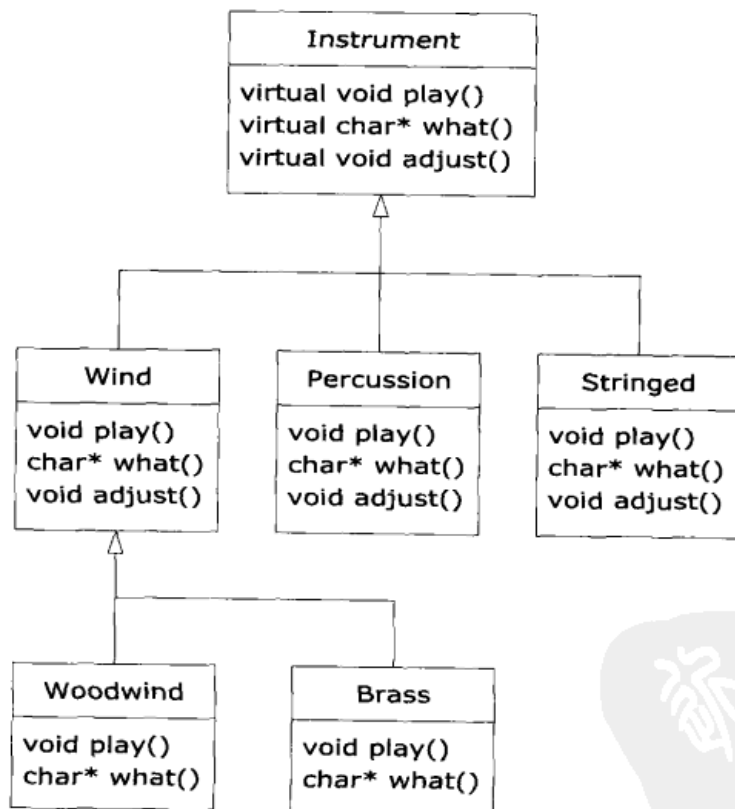
有些证据表明，C++中的规模和速度改进效果是在C的规模和速度的10%之内，并且常常更接近。能够得到更小的规模和更高速度的原因是C++可以有比用C更快的方法设计程序，而且设计的程序更小。

## 15.7 抽象基类和纯虚函数

在设计时，常常希望基类仅仅作为其派生类的一个接口。这就是说，仅想对基类进行向上类型转换，使用它的接口，而不希望用户实际地创建一个基类的对象。要做到这点，可以在基类中加入至少一个纯虚函数（*pure virtual function*），来使基类成为抽象（*abstract*）类。纯虚函数使用关键字**virtual**，并且在其后面加上**= 0**。如果某人试着生成一个抽象类的对象，编译器会制止他。这个工具允许生成特定的设计。

当继承一个抽象类时，必须实现所有的纯虚函数，否则继承出的类也将是一个抽象类。创建一个纯虚函数允许在接口中放置成员函数，而不一定要提供一段可能对这个函数毫无意义的代码。同时，纯虚函数要求继承出的类对它提供一个定义。

在所有的instrument的例子中，基类**Instrument**中的函数总是“哑”函数。如果调用这些函数，就会出错。这是因为，**Instrument**的目的是对所有从它派生出来的类创建公共接口。



建立公共接口的惟一原因是它能对于每个不同的子类有不同的表示。它建立一个基本的格式，用来确定什么是对于所有派生类是公共的——除此之外，别无用途。所以，把**Instrument**设计为抽象类就比较合适。当仅希望通过一个公共接口来操纵一组类，且这个公共接口不需要实现（或者不需要完全实现）时，可以创建一个抽象类。

如果有一个作用类似于抽象类的类（就像**Instrument**），则这个类的对象几乎总是没有意义的。也就是说，**Instrument**的含义只表示接口，不表示特例实现，所以创建一个

**Instrument**对象没有意义。我们也许想防止用户这样做，这可以通过让**Instrument**的所有虚函数打印出错信息而完成，但这种方法到运行时才能获得出错信息，并且要求用户进行可靠而详尽的测试。所以最好是在编译时就能发现这个问题。

下面是用于纯虚函数声明的语法：

```
virtual void f() = 0;
```

这样做，等于告诉编译器在VTABLE中为函数保留一个位置，但在这个特定位置中不放地址。只要有一个函数在类中被声明为纯虚函数，则VTABLE就是不完全的。

如果一个类的VTABLE是不完全的，当某人试图创建这个类的对象时，编译器做什么呢？它不能安全地创建一个纯抽象类的对象，所以如果试图创建一个纯抽象类的对象，编译器就发出一个出错信息。这样，编译器就保证了抽象类的纯洁性，它就不会被误用了。

下面是修改后的**Instrument4.cpp**，它使用了纯虚函数。因为这个类中全是纯虚函数，所以我们称之为纯抽象类 (*pure abstract class*)：

```
//: C15:Instrument5.cpp
// Pure abstract base classes
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
    // Pure virtual functions:
    virtual void play(note) const = 0;
    virtual char* what() const = 0;
    // Assume this will modify the object:
    virtual void adjust(int) = 0;
};
// Rest of the file is the same ...

class Wind : public Instrument {
public:
    void play(note) const {
        cout << "Wind::play" << endl;
    }
    char* what() const { return "Wind"; }
    void adjust(int) {}
};

class Percussion : public Instrument {
public:
    void play(note) const {
        cout << "Percussion::play" << endl;
    }
    char* what() const { return "Percussion"; }
    void adjust(int) {}
};

class Stringed : public Instrument {
public:
    void play(note) const {
        cout << "Stringed::play" << endl;
    }
};
```



```

    }
    char* what() const { return "Stringed"; }
    void adjust(int) {}
};

class Brass : public Wind {
public:
    void play(note) const {
        cout << "Brass::play" << endl;
    }
    char* what() const { return "Brass"; }
};

class Woodwind : public Wind {
public:
    void play(note) const {
        cout << "Woodwind::play" << endl;
    }
    char* what() const { return "Woodwind"; }
};

// Identical function from before:
void tune(Instrument& i) {
    // ...
    i.play(middleC);
}

// New function:
void f(Instrument& i) { i.adjust(1); }

int main() {
    Wind flute;
    Percussion drum;
    Stringed violin;
    Brass flugelhorn;
    Woodwind recorder;
    tune(flute);
    tune(drum);
    tune(violin);
    tune(flugelhorn);
    tune(recorder);
    f(flugelhorn);
} ///:~

```

纯虚函数是非常有用的，因为它们使得类有明显的抽象性，并告诉用户和编译器打算如何使用。

注意，纯虚函数禁止对抽象类的函数以传值方式调用。这也是防止对象切片 (*object slicing*) (这将会被简单地介绍) 的一种方法。通过抽象类，可以保证在向上类型转换期间总是使用指针或引用。

纯虚函数防止产生完全的VTABLE，但这并不意味着我们不希望对其他一些函数产生函数体。我们常常希望调用一个函数的基类版本，即便它是虚拟的。把公共代码放在尽可能靠近我们的类层次根的地方，这是很好的想法。这不仅节省了代码空间，而且使得改变的传播更加容易。

### 15.7.1 纯虚定义

在基类中，对纯虚函数提供定义是可能的。我们仍然告诉编译器不允许产生抽象基类的对象，而且如果要创建对象，则纯虚函数必须在派生类中定义。然而，我们可能希望一段公共代码，使一些或所有派生类定义都能调用，而不必在每个函数中重复这段代码。

正如下面的纯虚定义：

```
//: C15:PureVirtualDefinitions.cpp
// Pure virtual base definitions
#include <iostream>
using namespace std;

class Pet {
public:
    virtual void speak() const = 0;
    virtual void eat() const = 0;
    // Inline pure virtual definitions illegal:
    //! virtual void sleep() const = 0 {}
};

// OK, not defined inline
void Pet::eat() const {
    cout << "Pet::eat()" << endl;
}

void Pet::speak() const {
    cout << "Pet::speak()" << endl;
}

class Dog : public Pet {
public:
    // Use the common Pet code:
    void speak() const { Pet::speak(); }
    void eat() const { Pet::eat(); }
};

int main() {
    Dog simba; // Richard's dog
    simba.speak();
    simba.eat();
} ///:~
```

**Pet**的VTABLE表仍然空着，但在这个派生类中刚好有一个函数，可以通过名字调用它。

这个特点的另一个好处是，它允许我们实现从常规虚函数到纯虚函数的改变，而无须打乱已存在的代码。（这是一个处理不用重新定义虚函数的类的方法。）

## 15.8 继承和VTABLE

可以想象，当实现继承和重新定义一些虚函数时，会发生什么事情？编译器对新类创建一个新VTABLE表，并且插入新函数的地址，对于没有重新定义的虚函数使用基类函数的地址。无论如何，对于可被创建的每个对象（即它的类不含有纯虚函数），在VTABLE中总有一个函数地址的全集，所以绝对不能对不在其中的地址进行调用（否则结果将会是灾难性的）。

但是在派生（*derived*）类中继承或增加新的虚函数时会发生什么呢？下面有一个简单的例子：

```

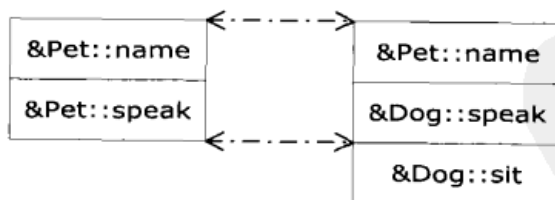
//: C15:AddingVirtuals.cpp
// Adding virtuals in derivation
#include <iostream>
#include <string>
using namespace std;
class Pet {
    string pname;
public:
    Pet(const string& petName) : pname(petName) {}
    virtual string name() const { return pname; }
    virtual string speak() const { return ""; }
};

class Dog : public Pet {
    string name;
public:
    Dog(const string& petName) : Pet(petName) {}
    // New virtual function in the Dog class:
    virtual string sit() const {
        return Pet::name() + " sits";
    }
    string speak() const { // Override
        return Pet::name() + " says 'Bark!'";
    }
};

int main() {
    Pet* p[] = {new Pet("generic"), new Dog("bob")};
    cout << "p[0]->speak() = "
          << p[0]->speak() << endl;
    cout << "p[1]->speak() = "
          << p[1]->speak() << endl;
    //! cout << "p[1]->sit() = "
    //! << p[1]->sit() << endl; // Illegal
} ///:~

```

类**Pet**中含有2个虚函数：**speak()**和**name()**，而在类**Dog**中又增加了第3个称为**sit()**的虚函数，并且重新定义了**speak()**的含义。下图有助于显示发生的事情。这是由编译器为**Pet**和**Dog**创建的VTABLE。



注意，编译器在**Dog**的VTABLE中把**speak()**的地址准确地映射到和**Pet**的VTABLE中同样的位置。类似地，如果类**Pug**从**Dog**中继承而来，则在它的VTABLE中**sit()**也将会被放置在和**Dog**的VTABLE中相同的位置。这是因为（正如通过汇编语言例子看到的）编译器产生的代码在VTABLE中使用一个简单的偏移来选择虚函数。不论对象属于哪个特殊的类，它的VTABLE都是以同样的方法设置，所以对虚函数的调用将总是使用同样的方法。

然而在这里，编译器只对指向基类对象的指针工作。而这个基类只有**speak()**和**name()**



函数，所以它就是编译器惟一允许调用的函数。那么，如果只有基类对象的指针，那么编译器怎么可能知道自己正在对**Dog**对象工作呢？这个指针可能指向其他一些没有**sit()**函数的类。在**VTABLE**中，可能有，也可能没有一些其他函数的地址，但无论何种情况，对这个**VTABLE**地址做虚函数调用都不是我们想要做的。所以编译器通过防止我们对只存在于派生类中的函数做虚函数调用来完成其工作。

有一些比较少见的情况，可能我们知道指针实际上指向哪一种特殊子类的对象。这时如果想调用只存在于这个子类中的函数，则必须类型转换这个指针。下面的语句可以消除由前面程序产生的出错信息：

```
((Dog*)p[1])->sit()
```

这里，我们碰巧知道**p[1]**指向**Dog**对象，但通常情况下我们并不知道。如果你的问题是必须知道所有对象的确切类型，那么我们应当重新考虑这个问题，因为我们可能在进行不正确的虚函数调用。然而对于有些情况，如果知道保存在一般容器中的所有对象的确切类型，会使我们的设计工作在最佳状态（或者没有选择）。这就是运行时类型辨认（*Run-Time Type Identification*, RTTI）问题。

RTTI是有关向下类型转换基类指针到派生类指针的问题（“向上”和“向下”是相对典型类图而言的，典型类图以基类为顶点）。向上类型转换是自动发生的，不需强制，因为它是绝对安全的。向下类型转换是不安全的，因为这里没有关于实际类型的编译时信息，所以必须准确地知道这个类实际上是什么类型。如果把它转换成错误的类型，就会出现麻烦。

在本章的后面将讨论RTTI，而且本书的第2卷中也有一章专门讨论这个主题。

### 15.8.1 对象切片

当多态地处理对象时，传地址与传值有明显的不同。所有在这里已经看到的例子和将会看到的例子都是传地址的，而不是传值的。这是因为地址都有相同的长度<sup>①</sup>，传递派生类（它通常稍大一些）对象的地址和传递基类（它通常更小一点）对象的地址是相同的。如前面所述，这是使用多态的目的，即让对基类对象操作的代码也能透明地操作派生类对象。

如果对一个对象进行向上类型转换，而不使用地址或引用，发生的事情将会使我们吃惊：这个对象被“切片”，直到剩下的是适合于目的的子对象。在下面例子中可以看到当一个对象被“切片”后发生了什么。

```
//: C15:ObjectSlicing.cpp
#include <iostream>
#include <string>
using namespace std;

class Pet {
    string pname;
public:
    Pet(const string& name) : pname(name) {}
    virtual string name() const { return pname; }
    virtual string description() const {
        return "This is " + pname;
    }
}
```

① 实际上，并不是所有机器上的指针都具有同样的长度。然而，在我们的讨论范围内，认为它们是相同的。

```
};

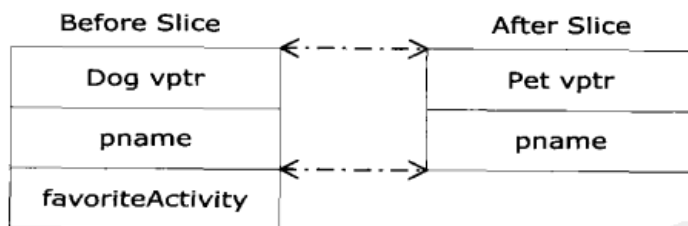
class Dog : public Pet {
    string favoriteActivity;
public:
    Dog(const string& name, const string& activity)
        : Pet(name), favoriteActivity(activity) {}
    string description() const {
        return Pet::name() + " likes to " +
            favoriteActivity;
    }
};

void describe(Pet p) { // Slices the object
    cout << p.description() << endl;
}

int main() {
    Pet p("Alfred");
    Dog d("Fluffy", "sleep");
    describe(p);
    describe(d);
} ///:~
```

函数 `describe()` 通过传值方式传递一个类型为 `Pet` 的对象。然后对于这个 `Pet` 对象调用虚函数 `description()`。我们可能希望第一次调用产生 “This is Alfred”，而第二次调用产生 “Fluffy likes to sleep”。实际上，两次调用都是调用了基类版本的 `description()`。

在这个程序中，发生了两件事情。第一，`describe()` 接受的是一个 `Pet` 对象（而不是指针或引用），所以 `describe()` 中的任何调用都将引起一个与 `Pet` 大小相同的对象压栈并在调用后清除。这意味着，如果一个由 `Pet` 派生来的类的对象被传给 `describe()`，则编译器会接受它，但只拷贝这个对象的对应于 `Pet` 的部分，切除这个对象的派生部分，如下图所示：



现在，我们可能对这个虚函数调用有这样的疑问：如果 `Dog::description()` 使用了 `Pet`（它仍存在）和 `Dog`（它不再存在，因为已被切掉），当调用它时，会发生什么呢？

其实我们已经从灾难中被解救出来，这个对象正安全地按值传递。这是因为派生类对象已经被强迫地变为基类对象，所以编译器知道这个对象的确切类型。另外，当按值传递时，`Pet` 对象的拷贝构造函数被调用，该构造函数初始化 `VPTR` 指向 `Pet` 的 `VTABLE`，并且只拷贝这个对象的 `Pet` 部分。这里没有显式的拷贝构造函数，所以编译器自动地生成一个。由于所有上述原因，因此这个对象在切片过程中真的变成了一个 `Pet` 对象。

对象切片实际上是当它拷贝到一个新的对象时，去掉原来对象的一部分，而不是像使用指针或引用那样简单地改变地址的内容。因此，不常使用对象向上类型转换，事实上，通常是要提防或防止这种操作。注意，在本例中，如果 `description()` 在基类中是一个纯虚函数

(这并不是毫无理由的，因为它在基类中实际上也并没有做什么事情)，因为编译器不会允许我们“创建”基类对象（这就是我们通过传值向上类型转换所发生的事情），所以它将阻止对对象进行“切片”。这可能会是纯虚函数最重要的作用：如果某人试着这么做，将通过生成一个编译错误来阻止对象切片。

## 15.9 重载和重新定义

在第14章中，我们看到重新定义一个基类中的重载函数将会隐藏所有该函数的其他基类版本。而当对虚函数进行这些操作时，情况会有点不同。考虑下面这个例子，它对第14章中的例子**NameHiding.cpp**进行了修改：

```

//: C15:NameHiding2.cpp
// Virtual functions restrict overloading
#include <iostream>
#include <string>
using namespace std;

class Base {
public:
    virtual int f() const {
        cout << "Base::f()\n";
        return 1;
    }
    virtual void f(string) const {}
    virtual void g() const {}
};

class Derived1 : public Base {
public:
    void g() const {}
};

class Derived2 : public Base {
public:
    // Overriding a virtual function:
    int f() const {
        cout << "Derived2::f()\n";
        return 2;
    }
};

class Derived3 : public Base {
public:
    // Cannot change return type:
    //! void f() const{ cout << "Derived3::f()\n";}
};

class Derived4 : public Base {
public:
    // Change argument list:
    int f(int) const {
        cout << "Derived4::f()\n";
        return 4;
    }
}

```



```
};

int main() {
    string s("hello");
    Derived1 d1;
    int x = d1.f();
    d1.f(s);
    Derived2 d2;
    x = d2.f();
    //! d2.f(s); // string version hidden
    Derived4 d4;
    x = d4.f(1);
    //! x = d4.f(); // f() version hidden
    //! d4.f(s); // string version hidden
    Base& br = d4; // Upcast
    //! br.f(1); // Derived version unavailable
    br.f(); // Base version available
    br.f(s); // Base version available
} ///:~
```

首先注意到，在**Derived3**中，编译器不允许我们改变重新定义过的函数的返回值（如果**f()**不是虚函数，则是允许的）。这是一个非常重要的限制，因为编译器必须保证我们能够多态地通过基类调用函数，并且如果基类希望**f()**返回一个**int**值，则**f()**的派生类版本必须保持约定，否则将会出问题。

在第14章中的规则仍将有效：如果重新定义了基类中的一个重载成员函数，则在派生类中其他重载函数将会被隐藏。这可由**main()**中测试**Derived4**的代码显示出来，即使**f()**的新版本实际上并没有重新定义一个已存在的虚函数的接口，**f()**的两个基类版本会被**f(int)**隐藏。然而，如果把**d4**向上类型转换到**Base**，则只有基类版本是可行的（因为基类约定允许），而派生类版本是不可行的（因为在基类中没有特定的方法）。

### 15.9.1 变量返回类型

上例的类**Derived3**显示了我们不能在重新定义过程中修改虚函数的返回类型。通常是这样的，但也有特例，我们可以稍稍修改返回类型。如果返回一个指向基类的指针或引用，则该函数的重新定义版本将会从基类返回的内容中返回一个指向派生类的指针或引用。例如：

```
//: C15:VariantReturn.cpp
// Returning a pointer or reference to a derived
// type during overriding
#include <iostream>
#include <string>
using namespace std;

class PetFood {
public:
    virtual string foodType() const = 0;
};

class Pet {
public:
    virtual string type() const = 0;
    virtual PetFood* eats() = 0;
```



```

};

class Bird : public Pet {
public:
    string type() const { return "Bird"; }
    class BirdFood : public PetFood {
    public:
        string foodType() const {
            return "Bird food";
        }
    };
    // Upcast to base type:
    PetFood* eats() { return &bf; }
private:
    BirdFood bf;
};

class Cat : public Pet {
public:
    string type() const { return "Cat"; }
    class CatFood : public PetFood {
    public:
        string foodType() const { return "Birds"; }
    };
    // Return exact type instead:
    CatFood* eats() { return &cf; }
private:
    CatFood cf;
};

int main() {
    Bird b;
    Cat c;
    Pet* p[] = { &b, &c, };
    for(int i = 0; i < sizeof p / sizeof *p; i++)
        cout << p[i]->type() << " eats "
            << p[i]->eats()->foodType() << endl;
    // Can return the exact type:
    Cat::CatFood* cf = c.eats();
    Bird::BirdFood* bf;
    // Cannot return the exact type:
    //! bf = b.eats();
    // Must downcast:
    bf = dynamic_cast<Bird::BirdFood*>(b.eats());
} ///:~

```

成员函数**Pet::eats()**返回一个指向**PetFood**的指针。在**Bird**中，完全按基类中的形式重载这个成员函数，并且包含了返回类型。也就是说，**Bird::eats()**把**BirdFood**向上类型转换到**PetFood**。

但在**Cat**中，**eats()**的返回类型是指向**CatFood**的指针，而**CatFood**是派生于**PetFood**的类。编译它的惟一原因是，返回类型是从基类函数的返回类型中继承而来的。这样，合约仍被遵守；**eats()**还是返回了一个**PetFood**指针。

如果考虑多态性的话，这看上去就并不是必需的。为什么不把所有的返回类型向上类型

转换为**PetFood\***，正如**Bird::eats()**所做的那样呢？这是个好建议，但在**main()**的结束部分，我们看到了不同之处：**Cat::eats()**可以返回**PetFood**的确切类型，而**Bird::eats()**的返回值必须被向下类型转换为确切的类型。

所以说，能返回确切的类型要更通用些，而且在自动地进行向上类型转换时不丢失特定的信息。然而，返回基类类型通常会解决我们的问题，所以这是一个特殊的功能。

## 15.10 虚函数和构造函数

当创建一个包含有虚函数的对象时，必须初始化它的VPTR以指向相应的VTABLE。这必须在对虚函数进行任何调用之前完成。正如我们可能猜到的，因为生成一个对象是构造函数的工作，所以设置VPTR也是构造函数的工作。编译器在构造函数的开头部分秘密地插入能初始化VPTR的代码。正如第14章所述，如果我们没有为一个类显式创建构造函数，则编译器会为我们生成构造函数。如果该类含有虚函数，则生成的构造函数将会包含相应的VPTR初始化代码。这有几个含义。

首先，这涉及效率。内联(**inline**)函数的作用是对小函数减少调用代价。如果C++不提供内联函数，则预处理器就可能被用来创建这些“宏”。然而，预处理器没有访问或类的概念，因此不能被用来创建成员函数宏。另外，有了由编译器插入的隐藏代码的构造函数，预处理宏根本不能工作。

当寻找效率漏洞时，我们必须明白，编译器正在插入隐藏代码到我们的构造函数中。这些隐藏代码不仅必须初始化VPTR，而且还必须检查**this**的值（以免**operator new**返回零）和调用基类构造函数。放在一起，这些代码可以影响我们认为是一个小内联函数的调用。特别是，构造函数的规模会抵消函数调用代价的减少。如果做大量的内联构造函数调用，代码长度就会增长，而在速度上没有任何好处。

当然，也许并不会立即把所有这些小构造函数都变成非内联，因为它们更容易写为内联构造函数。但是，当我们正在调整我们的代码时，记住，务必去掉这些内联构造函数。

### 15.10.1 构造函数调用次序

构造函数和虚函数的第二个有趣的方面涉及构造函数的调用顺序和在构造函数中虚函数调用的方法。

所有基类构造函数总是在继承类构造函数中被调用。这是有意义的，因为构造函数有一项专门的工作：确保对象被正确地建立。派生类只访问它自己的成员，而不访问基类的成员。只有基类构造函数能正确地初始化它自己的成员。因此，确保所有的构造函数被调用是很关键的，否则整个对象不会适当地被构造。这就是为什么编译器强制为派生类的每个部分调用构造函数的原因。如果不在构造函数初始化表达式表中显式地调用基类构造函数，它就调用默认构造函数。如果没有默认构造函数，编译器将报告出错。

构造函数调用的顺序是重要的。当继承时，必须知道基类的全部成员并能访问基类的任何**public**和**protected**成员。这意味着，当在派生类中时，必须能肯定基类的所有成员都是有效的。在通常的成员函数中，构造已经发生，所以这个对象的所有部分的成员都已经建立。然而，在构造函数内，必须想办法保证所有成员都已经建立。保证它的惟一方法是让基类构造函数首先被调用。这样，当在派生类构造函数中时，在基类中能访问的所有成员都已经被

初始化。在构造函数中，“知道所有成员对象是有效的”也是下面做法的原因：只要可能，我们应当在这个构造函数初始化表达式表中初始化所有的成员对象（即对象通过组合被置于类中）。只要遵从这个做法，我们就能保证初始化所有基类成员和当前对象的成员对象。

### 15.10.2 虚函数在构造函数中的行为

构造函数调用层次会导致一个有趣的两难选择。试想：如果我们在构造函数中并且调用了虚函数，那么会发生什么现象呢？在普通的成员函数中，我们可以想象所发生的情况——虚函数的调用是在运行时决定的，这是因为编译时这个对象并不能知道它是属于这个成员函数所在的那个类，还是属于由它派生出来的某个类。于是，我们也许会认为在构造函数中也会发生同样的事情。

然而，情况并非如此。对于在构造函数中调用一个虚函数的情况，被调用的只是这个函数的本地版本。也就是说，虚机制在构造函数中不工作。

这种行为有两个理由。在概念上，构造函数的工作是生成一个对象。在任何构造函数中，可能只是部分形成对象——我们只能知道基类已被初始化，但并不能知道哪个类是从这个基类继承来的。然而，虚函数在继承层次上是“向前”和“向外”进行调用。它可以调用在派生类中的函数。如果我们在构造函数中也这样做，那么我们所调用的函数可能操作还没有被初始化的成员，这将导致灾难的发生。

第二个理由是机械的。当一个构造函数被调用时，它做的首要的事情之一就是初始化它的VPTR。然而，它只能知道它属于“当前”类——即构造函数所在的类。于是它完全不知道这个对象是否是基于其他类的。当编译器为这个构造函数产生代码时，它是为这个类的构造函数产生代码——既不是为基类，也不是为它的派生类（因为类不知道谁继承它）。所以它使用的VPTR必须是对于这个类的VTABLE。而且，只要它是最后的构造函数调用，那么在这个对象的生命期内，VPTR将保持被初始化为指向这个VTABLE。但如果接着还有一个更晚派生的构造函数被调用，那么这个构造函数又将设置VPTR指向它的VTABLE，以此类推，直到最后的构造函数结束。VPTR的状态是由被最后调用的构造函数确定的。这就是为什么构造函数调用是按照从基类到最晚派生的类的顺序的另一个理由。

但是，当这一系列构造函数调用正发生时，每个构造函数都已经设置VPTR指向它自己的VTABLE。如果函数调用使用虚机制，它将只产生通过它自己的VTABLE的调用，而不是最后派生的VTABLE（所有构造函数被调用后才会有最后派生的VTABLE）。另外，许多编译器认识到，如果在构造函数中进行虚函数调用，应该使用早捆绑，因为它们知道晚捆绑将只对本地函数产生调用。无论哪种情况，在构造函数中调用虚函数都不能得到预期的结果。

### 15.11 析构函数和虚拟析构函数

构造函数是不能为虚函数的。但析构函数能够且常常必须是虚的。

构造函数有一项特殊工作，即一块一块地组合成一个对象。它首先调用基类构造函数，然后调用在继承顺序中的更晚派生的构造函数（同样，它也必须按此方法调用成员对象构造函数）。类似地，析构函数也有一项特殊工作，即它必须拆卸属于某层次类的对象。为了做这些工作，编译器生成代码来调用所有的析构函数，但它必须按照与构造函数调用相反的顺序。这就是，析构函数自最晚派生的类开始，并向上到基类。这是安全且合理的：当前的析构函

数一直知道基类成员仍是有效的。如果需要在析构函数中调用某一基类的成员函数，进行这样的操作是安全的。因此，析构函数能够对其自身进行清除，然后它调用下一个析构函数，该析构函数又将执行它的清除工作，以此类推。每个析构函数知道它所在类从哪一个类派生而来，但不知道从它派生出哪些类。

应当记住，构造函数和析构函数是类层次进行调用的惟一地方（因此，编译器自动地生成适当的类层次）。在所有其他函数中，只有这个函数会被调用（非基类版本），而无论它是虚的还是非虚的。同一个函数的基类版本在普通函数中被调用（无论它是虚的还是非虚的）的惟一方法是显式地调用这个函数。

通常，析构函数的执行是相当充分的。但是，如果想通过指向某个对象基类的指针操纵这个对象（也就是，通过它的一般接口操纵这个对象），会发生什么现象呢？这在面向对象的程序设计中确实很重要。当我们想`delete`在栈中已经用`new`创建的对象指针时，就会出现这个问题。如果这个指针是指向基类的，在`delete`期间，编译器只能知道调用这个析构函数的基类版本。这听起来很耳熟，虚函数被创建恰恰是为了解决同样的问题。幸运的是，就像除了构造函数以外的所有其他函数一样，析构函数可以是虚函数。

```
//: C15:VirtualDestructors.cpp
// Behavior of virtual vs. non-virtual destructor
#include <iostream>
using namespace std;
class Base1 {
public:
    ~Base1() { cout << "~Base1()\n"; }
};

class Derived1 : public Base1 {
public:
    ~Derived1() { cout << "~Derived1()\n"; }
};

class Base2 {
public:
    virtual ~Base2() { cout << "~Base2()\n"; }
};

class Derived2 : public Base2 {
public:
    ~Derived2() { cout << "~Derived2()\n"; }
};

int main() {
    Base1* bp = new Derived1; // Upcast
    delete bp;
    Base2* b2p = new Derived2; // Upcast
    delete b2p;
} ///:~
```

当运行这个程序时，将会看到`delete bp`只调用基类的析构函数。`delete b2p`调用了派生类的析构函数，然后调用了基类的析构函数。这正是我们所希望的。不把析构函数设为虚函数是一个隐匿的错误，因为它常常不会对程序有直接的影响。但要注意它不知不觉地引入存储器泄



漏（关闭程序时内存未释放）。同样，这样的析构操作还有可能掩盖发生的问题。

即使析构函数像构造函数一样，是“例外”函数，但析构函数可以是虚的，这是因为这个对象已经知道它是什么类型（而在构造期间则不然）。一旦对象已被构造，它的VPTR就已被初始化，所以能发生虚函数调用。

### 15.11.1 纯虚析构函数

尽管纯虚析构函数在标准C++中是合法的，但在使用时有一个额外的限制：必须为纯虚析构函数提供一个函数体。这看起来有点违反常规；如果它需要一个函数体，那它又如何称之为“纯”？但如果我们记得构造函数和析构函数是具有特别意义的操作，特别是如果我们记得在一个类层次中总是会调用所有的析构函数，就会有所体会。如果我们不对一个纯虚析构函数进行定义，在析构期间将会调用什么函数体呢？因此，编译器和链接程序强迫纯虚析构函数一定要有一个函数体，这是十分必要的。

如果它是纯虚的，而且不得不有一个函数体，那么它的价值是什么呢？我们可以看到纯虚析构函数和非纯虚析构函数之间惟一的不同之处在于纯虚析构函数使得基类是抽象类，所以不能创建一个基类的对象（虽然如果基类的任何其他函数是纯虚函数，也是具有同样的效果）。

然而，当从某个含有虚析构函数的类中继承出一个类，情况变得有点复杂。不像其他的纯虚函数，我们不要求在派生类中提供纯虚函数的定义。下面的编译和链接便是证明。

```
//: C15:UnAbstract.cpp
// Pure virtual destructors
// seem to behave strangely

class AbstractBase {
public:
    virtual ~AbstractBase() = 0;
};

AbstractBase::~~AbstractBase() {}

class Derived : public AbstractBase {};
// No overriding of destructor necessary?

int main() { Derived d; } ///:~
```

一般来说，如果在派生类中基类的纯虚函数（和所有其他纯虚函数）没有重新定义，则派生类将会成为抽象类。但这里，看起来好像并不是这样。然而，如果不进行析构函数定义，编译器将会自动地为每个类生成一个析构函数定义。那就是这里所发生的——基类的析构函数被重写（重新定义），因此编译器会提供定义并且派生类实际上不会成为抽象类。

这会产生一个有趣的问题：纯虚析构函数的目的是什么？它不像普通的纯虚函数，我们必须提供一个函数体。在派生类中，由于编译器为我们生成了析构函数，所以我们并非一定要提供一个定义。那么，常规的析构函数和纯析构函数的差别是什么呢？

当我们的类仅含有一个纯虚函数时，就会发现这个惟一的差别：析构函数。在这一点上，析构函数的纯虚性的惟一效果是阻止基类的实例化。如果有其他的纯虚函数，则它们会阻止基类的实例化，但如果没有那些纯虚函数，则纯虚析构函数将会执行这项操作。所以，当虚析构函数是十分必要时，则它是不是纯虚的就不是那么重要了。

运行下面的程序，可以看到在派生类版本之后，随着任何其他的析构函数，调用了纯虚函数体。

```
//: C15:PureVirtualDestructors.cpp
// Pure virtual destructors
// require a function body
#include <iostream>
using namespace std;

class Pet {
public:
    virtual ~Pet() = 0;
};

Pet::~Pet() {
    cout << "~Pet()" << endl;
}

class Dog : public Pet {
public:
    ~Dog() {
        cout << "~Dog()" << endl;
    }
};

int main() {
    Pet* p = new Dog; // Upcast
    delete p; // Virtual destructor call
} ///:~
```

作为一个准则，任何时候我们的类中都要有一个虚函数，我们应当立即增加一个虚析构函数（即使它什么也不做）。这样，我们保证在后面不会出现问题。

### 15.11.2 析构函数中的虚机制

在析构期间，有一些我们可能不希望马上发生的情况。如果正在一个普通的成员函数中，并且调用一个虚函数，则会使用晚捆绑机制来调用这个函数。而对于析构函数，这样不行，不论是虚的还是非虚的。在析构函数中，只有成员函数的“本地”版本被调用；虚机制被忽略。

```
//: C15:VirtualsInDestructors.cpp
// Virtual calls inside destructors
#include <iostream>
using namespace std;

class Base {
public:
    virtual ~Base() {
        cout << "Base()\n";
        f();
    }
    virtual void f() { cout << "Base::f()\n"; }
};

class Derived : public Base {
```



```

public:
    ~Derived() { cout << "~Derived()\n"; }
    void f() { cout << "Derived::f()\n"; }
};

int main() {
    Base* bp = new Derived; // Upcast
    delete bp;
} ///:~

```

在析构函数的调用中，**Derived::f()**没有被调用，即使**f()**是一个虚函数。

为什么是这样呢？假设在析构函数中使用虚机制，那么调用下面这样的虚函数是可能的：这个函数是在继承层次中比当前的析构函数“更靠外的”（更晚派生的）。但是，有一点要注意，析构函数从“外层”（从最晚派生的析构函数向基类析构函数）被调用。所以，实际上被调用的函数就可能操作在已被删除的对象上。因此，编译器决定在编译时只调用这个函数的“本地”版本。注意，对于构造函数也是如此（这在前面已讲到），但在构造函数的情况下，这样做是因为类型信息还不可用，然而在析构函数中，这样做是因为信息（也就是VPTR）虽存在，但不可靠。

### 15.11.3 创建基于对象的继承

本书中，在对容器类**Stack**和**Stash**的描述中，有一点是重复出现的，这就是“所有权问题”。负责对动态创建（使用**new**）的对象进行**delete**调用的称为“所有者”。在使用容器时的的问题是，它们需要足够的灵活性用来接收不同类型的对象。为了做到这一点，容器使用**void**指针，因此它们并不知道所包容对象的类型。删除一个**void**指针并不调用析构函数，所以容器并不负责清除它的对象。

在第14章的例子**InheritStack.cpp**中提出了一种解决办法，从**Stack**继承出一个仅可以接收和生成**string**指针的类。所以它知道它只包容了指向**string**对象的指针，因此它可以正确地删除它们。这是一个不错的解决办法，但是它要求我们要为想在容器中容纳的每一种类型都派生出一个新类。（虽然现在看起来有点冗余，但在第16章中介绍过模板后，它运行得相当不错。）

问题是我们希望容器可以容纳更多的类型，但我们不想使用**void**指针。另外一种解决方法是使用多态性，它通过强制容器内的所有对象从同一个基类继承而来。这就是说，容器容纳了具有同一基类的对象，并随后调用虚函数——特别地，我们可以调用虚析构函数来解决所有权问题。

这种解决方法使用单根继承（*singly-rooted hierarchy*）或基于对象的继承（*object-based hierarchy*）（这是因为继承的根类通常称为“对象”）。可以看到使用单根继承还有其他一些优点。事实上，除了C++，每种面向对象的语言都强制使用这样的体系——当创建一个类时，都会直接或间接地从一个公共基类中继承出它，这个基类是由该语言的创建者生成的。C++中认为，强制地使用这个公共基类会引起太多的开销，所以便没有使用它。然而，我们可以在自己的项目中选择是否使用它，在本书的第2卷中将进一步讨论这个主题。

为了解决所有权问题，可以创建一个相当简单的类**Object**作为基类，它仅包含一个虚析构函数。**Stack**于是可以容纳继承自**Object**的类。

```
///: C15:OStack.h
```

```

// Using a singly-rooted hierarchy
#ifndef OSTACK_H
#define OSTACK_H

class Object {
public:
    virtual ~Object() = 0;
};

// Required definition:
inline Object::~~Object() {}

class Stack {
    struct Link {
        Object* data;
        Link* next;
        Link(Object* dat, Link* nxt) :
            data(dat), next(nxt) {}
    } * head;
public:
    Stack() : head(0) {}
    ~Stack() {
        while(head)
            delete pop();
    }
    void push(Object* dat) {
        head = new Link(dat, head);
    }
    Object* peek() const {
        return head ? head->data : 0;
    }
    Object* pop() {
        if(head == 0) return 0;
        Object* result = head->data;
        Link* oldHead = head;
        head = head->next;
        delete oldHead;
        return result;
    }
};
#endif // OSTACK_H ///:~

```

通过把所有的东西放在头文件中来简化问题，纯虚析构函数（所要求的）的定义以内联形式置于头文件中，并且`pop()`也是内联的（对于内联形式来说，它可能太大了）。

**Link**对象现在是指向**Object**指针，而不是**void**指针，并且**Stack**也将仅仅接收和返回**Object**指针。现在，**Stack**更具有灵活性，因为它容纳了大量不同的类型，而且也可以消除被置于**Stack**中的任一对象。新的限制（在第16章中，当对这个问题运用模板时，将不具有这个限制）是置于**Stack**中的所有内容都必须继承自**Object**。如果新建一个类，这还是可行的，但如果已经有了一个类（例如**string**），并且希望把它置于**Stack**中，又会如何呢？这种情况下，新类必须具备**string**和**Object**的特点，即它必须继承自这两个类。这称之为多重继承（*multiple inheritance*），在本书第2卷（可从[www.BruceEckel.com](http://www.BruceEckel.com)处下载）中有一整章是关于这个主题的。当我们阅读该章时，将会看到多重继承是非常复杂的，应尽量少用这一功能。

然而，在这里，所有的一切都是很简单的，所以无需考虑多重继承的任何缺点。

```

//: C15:OStackTest.cpp
//{T} OStackTest.cpp
#include "OStack.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

// Use multiple inheritance. We want
// both a string and an Object:
class MyString: public string, public Object {
public:
    ~MyString() {
        cout << "deleting string: " << *this << endl;
    }
    MyString(string s) : string(s) {}
};

int main(int argc, char* argv[]) {
    requireArgs(argc, 1); // File name is argument
    ifstream in(argv[1]);
    assure(in, argv[1]);
    Stack textlines;
    string line;
    // Read file and store lines in the stack:
    while(getline(in, line))
        textlines.push(new MyString(line));
    // Pop some lines from the stack:
    MyString* s;
    for(int i = 0; i < 10; i++) {
        if((s=(MyString*)textlines.pop())==0) break;
        cout << *s << endl;
        delete s;
    }
    cout << "Letting the destructor do the rest:"
        << endl;
} ///:~

```

虽然这个代码段与**Stack**以前的测试程序版本很相似，但我们注意到仅有10个元素从栈中弹出，这意味着还保留了一些对象。因为**Stack**知道它包容了**Object**，并且析构函数可以正确地把它清除掉。因为**MyString**对象在它们被清除时打印信息，所以我们可从程序的输出中知道这一点。

创建包容**Object**的容器是一种合理的方法——如果使用单根继承（由于语言本身或需要的缘故，强制每个类继承自**Object**）。这时，保证一切都是一个**Object**，因此在使用容器时并不是十分复杂。然而，在C++中，不能期望这适用于每个类，所以如果有多重继承会出现问题。在第16章中会看到模板可以使用更简单、更灵巧的方法来处理这个问题。

## 15.12 运算符重载

就像对成员函数那样，我们可以使用**virtual**运算符。然而，因为我们可能对两个不知道

类型的对象进行操作，所以实现**virtual**运算符通常会很复杂。这通常用于处理数学部分（对于它们，我们常常重载运算符）。例如，对于一个处理矩阵、向量和标量的系统，这3个成分都是派生自**Math**类。

```
//: C15:OperatorPolymorphism.cpp
// Polymorphism with overloaded operators
#include <iostream>
using namespace std;

class Matrix;
class Scalar;
class Vector;

class Math {
public:
    virtual Math& operator*(Math& rv) = 0;
    virtual Math& multiply(Matrix*) = 0;
    virtual Math& multiply(Scalar*) = 0;
    virtual Math& multiply(Vector*) = 0;
    virtual ~Math() {}
};

class Matrix : public Math {
public:
    Math& operator*(Math& rv) {
        return rv.multiply(this); // 2nd dispatch
    }
    Math& multiply(Matrix*) {
        cout << "Matrix * Matrix" << endl;
        return *this;
    }
    Math& multiply(Scalar*) {
        cout << "Scalar * Matrix" << endl;
        return *this;
    }
    Math& multiply(Vector*) {
        cout << "Vector * Matrix" << endl;
        return *this;
    }
};

class Scalar : public Math {
public:
    Math& operator*(Math& rv) {
        return rv.multiply(this); // 2nd dispatch
    }
    Math& multiply(Matrix*) {
        cout << "Matrix * Scalar" << endl;
        return *this;
    }
    Math& multiply(Scalar*) {
        cout << "Scalar * Scalar" << endl;
        return *this;
    }
    Math& multiply(Vector*) {
```



```

        cout << "Vector * Scalar" << endl;
        return *this;
    }
};

class Vector : public Math {
public:
    Math& operator*(Math& rv) {
        return rv.multiply(this); // 2nd dispatch
    }
    Math& multiply(Matrix*) {
        cout << "Matrix * Vector" << endl;
        return *this;
    }
    Math& multiply(Scalar*) {
        cout << "Scalar * Vector" << endl;
        return *this;
    }
    Math& multiply(Vector*) {
        cout << "Vector * Vector" << endl;
        return *this;
    }
};

int main() {
    Matrix m; Vector v; Scalar s;
    Math* math[] = { &m, &v, &s };
    for(int i = 0; i < 3; i++)
        for(int j = 0; j < 3; j++) {
            Math& m1 = *math[i];
            Math& m2 = *math[j];
            m1 * m2;
        }
} ///:~

```

为了简单起见，这里仅重载了**operator\***。重载的目的是使任意两个**Math**对象相乘并且生成所需的结果——注意矩阵乘以向量和向量乘以矩阵是两个完全不同的操作。

**main()**中的问题在于，表达式**m1 \* m2**包含了两个向上类型转换的**Math**引用，因此不知道这两个对象的类型。一个虚函数仅能进行单一指派——即判定一个未知对象的类型。本例中所使用的判定两个对象类型的技术称之为多重指派 (*multiple dispatching*)，一个单一虚函数调用引起了第二个虚函数调用。在完成第二个调用时，已经得到了这两个对象的类型，于是可以执行正确的操作。我们开始时会有点不清楚，但如果多看些例子，就会理解的。这个主题在本书的第2卷（可从[www.BruceEckel.com](http://www.BruceEckel.com)处下载）的“设计风格”一章中有更深入的探讨。

### 15.13 向下类型转换

我们可能猜测，既然存在向上类型转换——在类层次中向上移动，那也应该存在可以向下移动的向下类型转换 (*downcasting*)。但是由于在一个继承层次上向上移动时，类总是集中于更一般的类，因此向上类型转换是容易的。这就是说，当进行向上类型转换时，总是清楚地派生自祖先类（典型地总是一个，除了多重继承的情况），而当向下类型转换时，通常会有多种选择让我们进行类型转换。更特殊些，**Circle**是**Shape**的一种类型（这是向上类型转换），但如果对一

个**Shape**进行向下类型转换，它可能会是**Circle**、**Square**、**Triangle**等。因此，对于安全地进行向下类型转换，就出现了两难的选择。（但更重要的是，要问问自己，为什么首先使用向下类型转换而不用多态性来自动地获取正确的类型。在本书的第2章中介绍了向下类型转换的避免。）

C++提供了一个特殊的称为**dynamic\_cast**的显式类型转换（*explicit cast*）（在第3章中介绍过），它就是一种安全类型向下类型转换（*type-safe downcast*）的操作。当使用**dynamic\_cast**来试着向下类型转换一个特定的类型，仅当类型转换是正确的并且是成功的时，返回值会是一个指向所需类型的指针，否则它将返回0来表示这并不是正确的类型。下面有一个小例子。

```
//: C15:DynamicCast.cpp
#include <iostream>
using namespace std;

class Pet { public: virtual ~Pet(){} };
class Dog : public Pet {};
class Cat : public Pet {};

int main() {
    Pet* b = new Cat; // Upcast
    // Try to cast it to Dog*:
    Dog* d1 = dynamic_cast<Dog*>(b);
    // Try to cast it to Cat*:
    Cat* d2 = dynamic_cast<Cat*>(b);
    cout << "d1 = " << (long)d1 << endl;
    cout << "d2 = " << (long)d2 << endl;
} ///:~
```

当使用**dynamic\_cast**时，必须对一个真正多态的层次进行操作——它含有虚函数——这因为**dynamic\_cast**使用了存储在VTABLE中的信息来判断实际的类型。这里，基类含有一个虚析构函数，这就足够了。**main()**中，一个**Cat**指针被向上类型转换到**Pet**，然后又试着向下类型转换到一个**Dog**指针和一个**Cat**指针。运行这个程序时，打印出这两个指针，可以看到不正确的向下类型转换返回了0值。当然，无论何时进行向下类型转换，我们都有责任进行检验以确保类型转换的返回值为非0。但我们不用确保指针要完全一样，这是因为通常在向上类型转换和向下类型转换时指针会进行调整（特别是在多重继承的情况下）。

**dynamic\_cast**运行时需要一点额外的开销；不多，但如果执行大量的**dynamic\_cast**（这时我们的程序设计就有严重的问题），就会影响性能。有时，在进行向下类型转换时，我们可以知道正在处理的是何种类型，这时使用**dynamic\_cast**产生的额外开销就没有必要，可以通过使用**static\_cast**来代替它。

```
//: C15:StaticHierarchyNavigation.cpp
// Navigating class hierarchies with static_cast
#include <iostream>
#include <typeinfo>
using namespace std;

class Shape { public: virtual ~Shape() {} };
class Circle : public Shape {};
class Square : public Shape {};
class Other {};
```



```

int main() {
    Circle c;
    Shape* s = &c; // Upcast: normal and OK
    // More explicit but unnecessary:
    s = static_cast<Shape*>(&c);
    // (Since upcasting is such a safe and common
    // operation, the cast becomes cluttering)
    Circle* cp = 0;
    Square* sp = 0;
    // Static Navigation of class hierarchies
    // requires extra type information:
    if(typeid(s) == typeid(cp)) // C++ RTTI
        cp = static_cast<Circle*>(s);
    if(typeid(s) == typeid(sp))
        sp = static_cast<Square*>(s);
    if(cp != 0)
        cout << "It's a circle!" << endl;
    if(sp != 0)
        cout << "It's a square!" << endl;
    // Static navigation is ONLY an efficiency hack;
    // dynamic_cast is always safer. However:
    // Other* op = static_cast<Other*>(s);
    // Conveniently gives an error message, while
    Other* op2 = (Other*)s;
    // does not
} ///:~

```

在这个程序中，使用了一个新的特征，本书第2卷会有一章完全介绍这一主题：C++的运行时类型识别（*Run-Time Type Information*, RTTI）机制。RTTI允许我们得到在进行向上类型转换时丢失的类型信息。**dynamic\_cast**实际上就是RTTI的一种形式。这里，**typeid**关键字（在头文件**<typeinfo>**中声明）用来检测指针的类型。可以看到，向上类型转换的**Shape**指针的类型相继与**Circle**指针和**Square**指针相比较，来判断它们是否匹配。RTTI的内容远远不止**typeid**，我们也可以想象它能够通过虚函数简单合理地实现我们自己的类型信息系统。

程序创建了一个**Circle**对象，它的地址被向上类型转换为**Shape**指针；第二个表达式显示了我们如何使用**static\_cast**来进行更加显式地向上类型转换。然而，由于向上类型转换总是安全的并且是通用的，因此我认为用一个显式类型转换来进行向上类型转换将是混乱和没有必要的。

RTTI用于判定类型，**static\_cast**用于执行向下类型转换。但要注意，在这个设计中，处理效率同使用**dynamic\_cast**是一样的，并且客户程序员必须进行检测来发现那些实际成功的类型转换。我们希望在不使用**dynamic\_cast**而使用**static\_cast**之前，有一个比上面例子更加确定的环境（并且在使用**dynamic\_cast**之前，我们希望可以再一次仔细地检查我们的设计）。

如果类层次中没有虚函数（这是一个有问题的设计），或者如果有其他的需要，要求我们安全地进行向下类型转换，与使用**dynamic\_cast**相比，静态地执行向下类型转换会稍微快一点。另外，**static\_cast**不允许类型转换到该类层次的外面，而传统的类型转换是允许的，所以它们会更安全。但是静态地浏览类层次总是有风险的，所以除非特殊情况，我们一般使用**dynamic\_cast**。

## 15.14 小结

多态性在C++中用虚函数实现，它意味着“具有不同的形式”。在面向对象的程序设计中，

有相同的功能（即基类中的公共接口）和使用这个功能的不同的形式：虚函数的不同版本。

在本章中，我们已经看到，不用数据抽象和继承，理解甚至创建一个多态的例子，是不可能的。多态是不能独立看待的特征（例如像**const**或**switch**这样的语句），必须协同工作，它是类关系大家庭中的一部分。人们常常被C++的其他非面向对象的特征（例如重载和默认参数）所混淆，它们有时被作为面向对象的特征描述。不要被迷惑，如果它们没有进行晚捆绑，就没有多态性。

为了在程序中有效地使用多态等面向对象的技术，不能只知道让程序包含单个类的成员和消息，而且还应知道类的共性和它们之间的关系。虽然这需要很大的努力，但这是值得的，因为将得到更快的程序开发和更好的代码组织、可扩充的程序和更容易维护的代码。

多态性完善了语言的面向对象特征，但在C++中，还有两个更重要的特征：模板（第16章的内容，并且在第2卷中有更为详细介绍）和异常处理（在第2卷中介绍）。就像面向对象的其他特征（抽象数据类型、继承和多态）一样，这些特征使我们的编程能力有很大的提高。

## 15.15 练习

部分练习题的答案可以在本书的电子文档“*Annotated Solution Guide for Thinking in C++*”中找到，只需支付很少的费用就可以从<http://www.BruceEckel.com>得到这个电子文档。

- 15-1 创建一个非常简单的“shape”层次：基类称为**Shape**，派生类称为**Circle**、**Square**和**Triangle**。在基类中定义一个虚函数**draw()**，再在这些派生类中重定义它。在堆中创建**Shape**对象，并且建立一个指向这些**Shape**对象的指针数组（这样就形成了指针向上类型转换）。并且通过基类指针调用**draw()**，检验虚函数的行为。如果调试器支持，就用单步执行这个例子。
- 15-2 修改练习1，使得**draw()**是纯虚函数。尝试创建一个类型为**Shape**的对象。并试着在构造函数内调用这个纯虚函数，看看结果如何。保留它的纯虚性，对**draw()**进行定义。
- 15-3 在练习2的基础上进一步，创建一个通过传值方式接收**Shape**对象参数的函数，并试着向上类型转换一个派生类对象作为参数。看看结果如何。通过把参数设为**Shape**对象的引用来修改这个函数。
- 15-4 修改C14:Combined.cpp，把基类中的**f()**设为虚函数。在**main()**中执行向上类型转换并且调用虚函数。
- 15-5 通过增加一个虚函数**prepare()**来修改Instrument3.cpp。在**tune()**中调用**prepare()**。
- 15-6 创建一个含有**Rodent**类的继承层次，它包括**Mouse**、**Gerbil**、**Hamster**等。在基类中提供对所有**Rodent**都适用的方法，并根据**Rodent**的特定类型，在派生类中执行不同的行为。创建一个**Rodent**指针数组，使它们指向**Rodent**不同的特定类型，并且调用基类中的方法，看看结果如何。
- 15-7 修改练习6，用**vector<Rodent\*>**来代替指针数组。确保内存可以被正确地清除掉。
- 15-8 根据前面的**Rodent**类层次，从**Hamster**中继承出**BlueHamster**（是的，当我还是小孩时，我就有这么一只鼠），重新定义基类中的方法，并显示调用基类方法的代码不需要进行修改就可以在新类中使用。
- 15-9 在前面的**Rodent**类层次中，增加一个虚析构函数，并且使用**new**创建一个**Hamster**对象，向上类型转换成为一个**Rodent\***，然后**delete**该指针，显示它并不能调用层次中所有的

析构函数。把析构函数改为虚函数，显示这样就可以正确地调用所有的析构函数。

- 15-10 在前面的**Rodent**类层次中，修改**Rodent**，使它成为一个纯抽象基类。
- 15-11 使用基类**Aircraft**和它的不同的派生类，创建一个空中交通系统。使用**vector<Aircraft\*>**建立类**Tower**，给在它控制下的不同飞行器发送适当的信息。
- 15-12 通过从**Plant**中继承各种类型来建立一个温室的模型，并且在该温室中创建可以照看植物的机制。
- 15-13 在**Early.cpp**中，使**Pet**成为一个纯抽象基类。
- 15-14 在**AddingVirtuals.cpp**中，把**Pet**所有的成员函数改为纯虚函数，并对**name()**进行定义。使用**name()**的基类定义，对**Dog**进行必要的修改。
- 15-15 写出一个小程序以显示在普通成员函数中调用虚函数和在构造函数中调用虚函数的不同。这个程序应当表明两种调用会产生不同的结果。
- 15-16 通过从**Derived**中继承出一个类并且重新定义它的**f()**和析构函数来修改**VirtualsIn-Destructors.CPP**。在**main()**中，向上类型转换我们的新类，然后**delete**它。
- 15-17 在练习16的基础上，在每一个析构函数中增加对函数**f()**的调用。解释运行的结果。
- 15-18 创建含有一个数据成员的和含用另一个数据成员的派生类。编写一个非成员函数，它通过传值方式接收一个基类的对象，并且使用**sizeof**打印出该对象的大小。在**main()**中创建一个派生类的对象，打印出它的大小，然后调用我们的函数。解释运行的结果。
- 15-19 创建一个虚函数调用的简单例子，并且输出其汇编代码。找出虚函数调用的汇编代码，跟踪运行并解释这些代码。
- 15-20 编写一个类，含有一个虚函数和一个非虚函数。继承出一个新类，并生成该类的对象，然后向上类型转换为基类的指针。使用**<ctime>**中的**clock()**函数（需要在本地C库指南中找到它）来测出虚函数调用和非虚函数调用的区别。为了看到区别，需要在时间循环中对每个函数进行多次调用。
- 15-21 通过在**CLASS**宏的基类中增加一个虚函数（使它打印些信息）并且把析构函数改为虚函数来修改**C14:Order.cpp**。生成不同子类的对象，然后把它们向上类型转换为基类对象。检验虚操作的运行以及发生的适当的构造操作和析构操作。
- 15-22 编写一个含有3个重载虚函数的类。在新建类中继承出一个新类，并且重新定义其中一个函数。生成派生类的一个对象。我们是否可以通过派生类对象调用所有的基类函数呢？把该对象的地址向上类型转换为基类对象。我们是否可以通过此基类对象调用所有的3个函数呢？删去在派生类中所做的重写定义。现在我们又是否可以通过派生类对象调用所有的基类函数呢？
- 15-23 修改**VariantReturn.cpp**，显示它的行为可以使用引用和指针来进行工作。
- 15-24 在**Early.cpp**中，如何才能分辨出编译器的调用是使用了早捆绑还是晚捆绑？判断我们自己的编译器的调用属于哪种情况？
- 15-25 创建一个基类，含有一个**clone()**函数，它返回指向当前对象拷贝的指针。派生出两个子类，同时重新定义**clone()**，它返回它们各自类型拷贝的指针。在**main()**中，生成并且向上类型转换两个派生类型的对象，然后分别调用它们的**clone()**，并检验所克隆的拷贝是正确的子类型。试验我们的**clone()**函数，使得返回的类型是基类，再试着返回准确的派生类型。我们能否考虑到后一种方法所必需的环境？

- 15-26 通过创建自己的类，然后对它和**Object**进行多重继承，生成的对象置于**Stack**中来修改**OStackTest.cpp**。在**main()**中测试我们的类。
- 15-27 在**OperatorPolymorphism.cpp**中增加一个类**Tensor**。
- 15-28 (中级)创建一个不带数据成员和构造函数而只有一个虚函数的基类**X**，从**X**继承出类**Y**，它没有显式的构造函数。产生汇编代码并检验它，以确定**X**的构造函数是否被创建和调用，如果是的，这些代码做什么？解释我们的发现。**X**没有默认的构造函数，但是为什么编译器不报告出错？
- 15-29 (中级)修改练习28，为这两个类创建构造函数，让每个构造函数调用一个虚函数。产生汇编代码。确定在每个构造函数内**VPTR**在何处被赋值。在构造函数内编译器使用虚函数机制吗？确定为什么这些函数的本地版本仍被调用。
- 15-30 (高级)如果对象的参数为传值方式传递的函数调用不用早捆绑，则虚调用可能会访问不存在的部分。这可能吗？编写一些代码强制进行虚调用，看看是否会引起冲突。解释这个行为，检验当对象以传值方式传递时会发生什么现象。
- 15-31 (高级)通过我们处理器的汇编语言信息或者其他技术，找出简单调用所需的时间数及虚函数调用所需的时间数，从而得出虚函数调用需要多出多少时间。
- 15-32 确定执行时**VPTR**的**Sizeof**。现在对两个含有虚函数的类进行多重继承。在派生类中可以得到一个还是两个**VPTR**？
- 15-33 创建一个含有数据成员和虚函数的类。编写一个监视我们类对象的内存的函数，它打印出变化的部分。要做到这一点，我们需要进行试验并且不断地找出对象中**VPTR**的所在位置。
- 15-34 假设不存在虚函数，修改**Instrument4.cpp**，使得它使用**dynamic\_cast**来代替虚函数调用。解释为什么这不是一个好的方法。
- 15-35 修改**StaticHierarchyNavigation.cpp**，不使用C++ RTTI，而是通过基类中的虚函数**whatAmI()**和**enum type { Circles, Squares }**，来创建我们自己的RTTI。
- 15-36 在第12章的**PointerToMemberOperator.cpp**中，显示即使重载了**operator->\***，多态性依旧适用于成员指针。



## 模板介绍

继承和组合提供了重用对象代码的方法，而C++的模板特征提供了重用源代码的方法。

虽然 C++模板是通用的程序设计工具，但当它们引入了C++后，似乎就不再鼓励使用基于对象的容器类层次结构（在第15章的最后论证）了。例如，标准C++容器类和算法（在本书的第2卷中有两章对此问题进行解释，可以从[www.BruceEckel.com](http://www.BruceEckel.com)下载本书的第2卷。）是完全应用模板完成的，对程序员来说相对易于使用。

本章不仅阐述模板的基础，而且还介绍容器，它是面向对象程序设计的基本构件，几乎可以完全通过标准C++库中的容器实现。可以看到，本书使用的容器的例子——**Stash** 和 **Stack**——刚好适合于学习容器。在本章中，增加了迭代器（*iterator*）的概念。虽然容器是模板使用的理想的例子，但是在第2卷中（其中有一章是专门讨论高级模板）将会学到模板的许多别的用法。

### 16.1 容器

假定想创建一个栈，正如全书所做的这样。为了简单，这个栈类只存放**int**类型的值。

```
//: C16:IntStack.cpp
// Simple integer stack
//{L} fibonacci
#include "fibonacci.h"
#include "../require.h"
#include <iostream>
using namespace std;

class IntStack {
    enum { ssize = 100 };
    int stack[ssize];
    int top;
public:
    IntStack() : top(0) {}
    void push(int i) {
        require(top < ssize, "Too many push()es");
        stack[top++] = i;
    }
    int pop() {
        require(top > 0, "Too many pop()s");
        return stack[--top];
    }
};

int main() {
    IntStack is;
    // Add some Fibonacci numbers, for interest:
```



```

    for(int i = 0; i < 20; i++)
        is.push(fibonacci(i));
    // Pop & print them:
    for(int k = 0; k < 20; k++)
        cout << is.pop() << endl;
} ///:~

```

类**IntStack**是最为常见的下推栈的例子。为了简化，此处栈的尺寸是固定的，但是也可以对其进行修改，使得它能通过在堆中分配内存而自动扩展，如同在本书中到处被考查的**Stack**类一样。

**main()**向这个栈添加一些整数，然后再弹出它们。为了让这个例子更有趣，这些整数用**fibonacci()**函数生成，它生成传统的兔子繁殖数。下面是声明这个函数的头文件。

```

//: C16: fibonacci.h
// Fibonacci number generator
int fibonacci(int n); ///:~

```

下面是实现：

```

//: C16: fibonacci.cpp {0}
#include "../require.h"

int fibonacci(int n) {
    const int sz = 100;
    require(n < sz);
    static int f[sz]; // Initialized to zero
    f[0] = f[1] = 1;
    // Scan for unfilled array elements:
    int i;
    for(i = 0; i < sz; i++)
        if(f[i] == 0) break;
    while(i <= n) {
        f[i] = f[i-1] + f[i-2];
        i++;
    }
    return f[n];
} ///:~

```

这是一个相当有效的实现，因为它绝不会多次生成这些数。它使用**int**的**static**数组，编译器将这个**static**数组初始化为零。第一个**for**循环把下标*i*移到第一个数组元素为零的地方，然后**while**循环向这个数组添加斐波纳契数，直到期望的元素达到。但是注意，如果经过元素*n*的斐波纳契数（Fibonacci number）都已经被初始化，则完全跳过这个**while**循环。

### 16.1.1 容器的需求

很明显，一个整数栈不是一个重要的工具。容器类的真正需求是在堆上使用**new**创建对象和使用**delete**销毁对象的时候体现的。在一般程序设计问题中，程序员在编写程序时并不知道将来需要创建多少个对象。例如在设计空中交通指挥系统时不应限制这个系统能处理的飞机数目。我们不希望由于实际飞机的数目超过设计值而导致这个系统失败。在计算机辅助设计系统中，可以处理许多造型，只有用户能够（在运行时）确定到底需要多少造型。我们一旦注意到上述问题，便可以在程序开发中发现许多这样的例子。

依赖虚拟存储去处理“存储器管理”的C程序员常常发现**new**、**delete**和容器类的思想的混乱。表面上看，创建一个足够大的能包括任何可能需求的巨型全局数组是可行的。这可能不需要太多思考（或者并不需要弄清楚**malloc()**和**free()**），但是这样的程序接口性能较差，而且暗藏着难以捕捉的错误。

另外，如果我们创建一个巨型的C++对象的全局数组，那么构造函数和析构函数的开销会使系统效率显著地下降。C++中有更好的解决方法：用**new**创建需要的对象，将其指针放入容器中，待实际使用时将其取出并进行处理。用这种方法，所创建的只是确实需要的对象。通常，在启动程序时没有可用的初始化条件。**new**允许等待，直到在环境中相关事件发生后，再实际地创建这个对象。

在大多数情况下，应当创建用来存放感兴趣对象指针的容器。应当用**new**创建这些对象，然后把结果指针放在容器中（在这个过程中这是向上类型转换），当需要用到这些对象时再将指针从容器中取出。这项技术使得程序更具灵活性和一般性。

## 16.2 模板综述

现在出现了一个问题。**IntStack**可存放整数，但是也可能希望有一个栈可存放造型、航班、植物等数据对象。用强调重用性的语言每次从头重新开发代码，不是一个明智的办法。应该有更好的方法。

有3种源代码重用的方法：C方法，这里列出是为了对照；对C++产生过重大影响的Smalltalk方法；C++的模板方法。

### (1) C方法

毫无疑问，应该摒弃C方法，这是由于它表现繁琐、易发生错误、缺乏美感。用这种方法，需要拷贝**Stack**的源码并对其进行手工修改，这样就会引进新的错误。这是非常低效的技术。

### (2) Smalltalk 方法

Smalltalk（以及之后的Java）方法是通过继承来实现代码重用的，既简单又直观。为此，每个容器类包含通用的基类**Object**的项目（类似于第15章最后的例子）。Smalltalk的基类库十分重要，完全不需要从头创建类。相反，创建一个新类必须从已有类中继承，不能随意创建。可以从类库中选择功能和需求尽可能接近的一个已有类作为父类，并在对父类的继承中加以修正从而创建一个新类。很明显，这种方法由于可以减少我们的工作量，因而提高了我们的效率（这也说明了为什么需要花大量的时间去学习Smalltalk类库才能成为熟练的Smalltalk程序员）。

但是，这也意味着Smalltalk的所有类都是单个继承树的一部分。当创建新类时必须继承树的某一枝。树的大部分已经存在（它是Smalltalk的类库），树的根称为**Object**——这是每个Smalltalk容器所包含的同一个类。

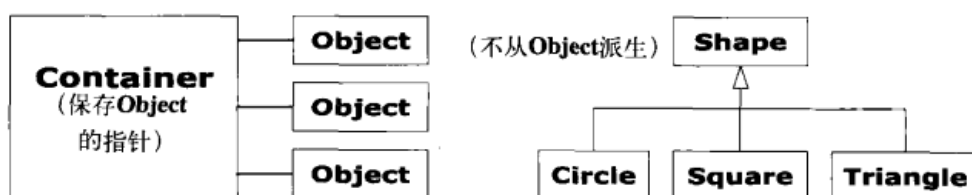
这是一种单纯的技巧，因为Smalltalk（和Java<sup>①</sup>）类层次上的任何类都源于**Object**的派生，所以任何容器可容纳任何类（包括容器本身）。这种基于通用的基类（常称为**Object**，在Java中也有类似情况）的单树形层次类型称为“基于对象的层次结构”。我们可能听说过这个概念，并猜想这是另一个OOP的基本概念，就像“多态性”一样。但实际上，这仅仅意味着以**Object**（或相近的名称）为根的树形类结构和包含**Object**的容器类。

① 在Java中，基本数据类型是一个例外，出于效率的考虑，这里有一些非**Object**类型。



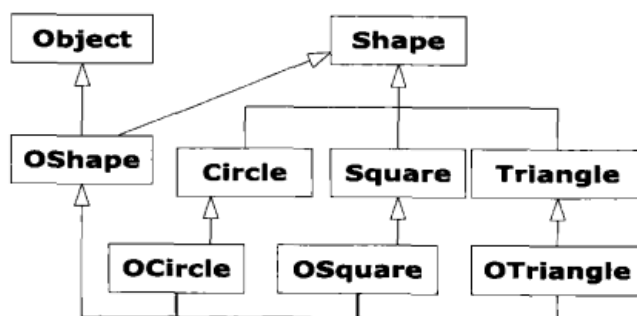
因为Smalltalk类库的发展史比C++更长久，且早期的C++编译器没有容器类库，所以C++能将Smalltalk类库的良好思想加以借鉴。这种借鉴出现在早期的C++实现中<sup>①</sup>，由于它表现为一个有效的代码实体，因此许多人开始使用它，但在使用容器类的过程中发现了一个问题。

该问题在于，在Smalltalk（和我所知道的许多其他OOP语言）中，所有的类都自动地从单个层次结构中派生而来，但在C++中则不行。我们可能本来已经拥有了完善的基于对象的层次结构以及它的容器类，而且还可能从其他不用这种层次结构的供应商那里购买到一组类，如形体类、航班类等（为了使用层次结构而增加了开销，这是C程序员不愿意做的事情）。我们如何把一个单独的类树插入到我们的基于对象的层次结构中的容器类之中呢？这个问题如下所示：



因为C++支持多个独立层次结构，所以Smalltalk的“基于对象的层次结构”在此不适用。

解决方案似乎是明显的。如果我们有许多继承层次结构，就应当能从多个类继承：多重继承可以解决上述问题。所以我们应该按下述的方法去实施（一个类似的例子已在第15章结尾处给出）：



现在，**OShape**具有了**Shape**的特点和行为，但它也是从**Object**派生而来的，所以可将其置于**Container**内。额外的继承也必然进入了**OCircle**和**OSquare**等，这样这些类才能向上类型转换为**OShape**，并因而保持正确的行为。我们可以看到，事情正在迅速变得混乱。

编译器供应商发明了他们自己的基于对象的容器类层次结构，并将它们加入到他们的编译系统中，这些层次结构中的大多数可以用模板版本替代。我们可以对多重继承是否可以解决大多数编程问题进行争论，但是在本书的第2卷中将会看到，除某些特殊情况外，它的复杂性是可以很好避免的。

### 16.2.1 模板方法

尽管具有多重继承的基于对象的层次结构在概念上是直观的，但是它在实践上较为困难。

<sup>①</sup> OOPS库，由Keith Gorlen在NIH时创建。

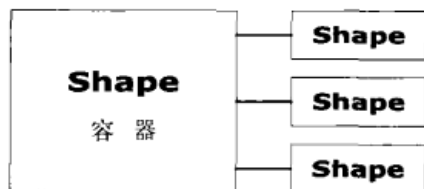


在Stroustrup的最初著作<sup>①</sup>中阐述了替换基于对象层次的一种更可取的选择。创造容器类作为参数化类型的大型预处理宏，这些参数能替代为所希望的类型。当我们打算创建一个容器存放某个特别类型时，应当使用一对宏调用。

不幸的是，这种方法在当时被所有已有的Smalltalk文献和程序设计经验弄混淆了，加之它确实有点难处理，所以当时基本上没有什么人用它。

在此期间，Stroustrup和贝尔实验室的C++小组对原先的宏方法进行了修正，对其进行了简化并将它从预处理器域移入到编译器中。这种新的代码替换装置被称为模板<sup>②</sup>，而且它表现了完全不同的代码重用方法：模板对源代码进行重用，而不是通过继承和组合重用目标代码。容器不再存放称为Object的通用基类，而是存放一个未指明的参数。当用户使用模板时，参数由编译器（*by the compiler*）来替换，这非常像原来的宏方法，但却更清晰、更容易使用。

现在，可以不必在使用容器类时担忧继承和组合了，可以采用容器的模板并且为具体问题复制出特定的版本，就像下图所示：



编译器会为我们做这些工作，而我们最终是以所需要的容器去做我们的工作，而不是用那些令人头疼的继承层次。在C++中，模板实现了参数化类型（*parameterized type*）的概念。模板方法的另一个优点是，使对继承不熟悉、不适应的新程序员也能正确地使用密封的容器类（就像在本书中对**vector**所做的那样）。

### 16.3 模板语法

**template**这个关键字会告诉编译器，随后的类定义将操作一个或更多未指明的类型。当由这个模板产生实际类代码时，必须指定这些类型以使编译器能够替换它们。

下面是一个说明模板语法的小例子，它产生一个带有越界检查的数组。

```

//: C16:Array.cpp
#include "../require.h"
#include <iostream>
using namespace std;

template<class T>
class Array {
    enum { size = 100 };
    T A[size];
public:
    T& operator[](int index) {
        require(index >= 0 && index < size,
            "Index out of range");
    }
};
  
```

① 《C++程序设计语言》（*The C++ Programming Language*），由Bjarne Stroustrup著（第1版，Addison-Wesley公司1986年出版）。

② 模板的思想类似于ADA的泛型（generic）。

```

        return A[index];
    }
};

int main() {
    Array<int> ia;
    Array<float> fa;
    for(int i = 0; i < 20; i++) {
        ia[i] = i * i;
        fa[i] = float(i) * 1.414;
    }
    for(int j = 0; j < 20; j++)
        cout << j << ": " << ia[j]
              << ", " << fa[j] << endl;
} ///:~

```

它看上去像一个普通的类，除了下面一行以外：

```
template<class T>
```

这里**T**是替换参数，它代表一个类型名称。在容器类中，它将出现在那些原本由某一特定类型出现的地方。

在**Array**中，其元素的插入和取出都用相同的函数——即重载的**operator[]**来实现。它返回一个引用，因此可被用于等号的两边（即，可以是左值也可以是右值）。注意，当下标值越界时，用**require()**函数输出提示信息。因为**operator[]**是内联的，所以用这种方法来保证不发生数组下标越界现象，随后在提交代码时去掉**require()**。

在**main()**中，我们看到可以非常容易地创建包含不同类型的**Array**。代码如下：

```

Array<int> ia;
Array<float> fa;

```

这时，编译器两次扩展了**Array**模板 [这被称为实例化 (*instantiation*) ]，创建两个新的生成类 (*generated class*)，可以把它们看做**Array\_int**和**Array\_float**（不同的编译器对名称有不同的修饰方法）。这些类就像手工创建的一样，只是这里是当定义了对象**ia**和**fa**后由编译器来创建这些类。我们还会注意到，编译器避免了或者连接器合并了类的重复定义。

### 16.3.1 非内联函数定义

当然，有时我们希望有非内联成员函数的定义。这时编译器需要在成员函数定义之前看到**template**声明。下面在前述例子的基础上加以修正来说明非内联函数的定义。

```

//: C16:Array2.cpp
// Non-inline template definition
#include "../require.h"

template<class T>
class Array {
    enum { size = 100 };
    T A[size];
public:
    T& operator[](int index);
};

```

```

template<class T>
T& Array<T>::operator[](int index) {
    require(index >= 0 && index < size,
        "Index out of range");
    return A[index];
}

int main() {
    Array<float> fa;
    fa[0] = 1.414;
} ///:~

```

注意在引用模板的类名的地方，必须伴有该模板的参数列表，例如在**Array<T>::operator[]**中。可以想象，在内部，使用模板参数列表中的参数修饰类名，以便为每一个模板实例产生惟一的类名标识符。

#### 16.3.1.1 头文件

即使是在创建非内联函数定义时，我们还是通常想把模板的所有声明和定义都放入一个头文件中。这似乎违背了通常的头文件规则：“不要放置分配存储空间任何东西”（这条规则是为了防止在连接期间的多重定义错误），但模板定义很特殊。在**template<...>**之后的任何东西都意味着编译器在当时不为它分配存储空间，而是一直处于等待状态直到被一个模板示例告知。在编译器和连接器中有机能能去掉同一模板的多重定义。所以为了使用方便，几乎总是在头文件中放置全部的模板声明和定义。

有时，也可能为了满足特殊的需要（例如，强制模板示例仅存在于单个的Windows **dll**文件中）而要在一个独立的**cpp**文件中放置模板的定义。大多数编译器有一些机制允许这么做；我们将必须检查我们的特定编译器的说明文档以便使用它。

有些人认为，在实现中将所有源代码放在头文件中，如果有人从我们这里买到库，则他们就有条件盗窃和修改代码。这可能是一个问题，但它依赖于我们看待这个问题的方法：他们买的是产品还是服务？如果是产品，我们就必须为保护它做一些事情，或许我们不想给出源代码，而只给出编译过的代码。但是许多人把软件看做服务，甚至是预约服务。消费者想要我们的专门技术，想要我们继续维护这段可重用的代码，所以他们没有必要这样做，因此他们可以集中精力做他们的事情。我个人认为，大多数消费者将我们看做有价值的资源，不希望危害他们与我们之间的关系。至于少数想盗窃而不是购买或做独创工作的人，他们大概无论如何也不能与我们相处。

#### 16.3.2 作为模板的IntStack

下面是来自**IntStack.cpp**的容器和迭代器，是作为一般的容器类使用模板来实现的：

```

//: C16:StackTemplate.h
// Simple stack template
#ifndef STACKTEMPLATE_H
#define STACKTEMPLATE_H
#include "../require.h"

template<class T>
class StackTemplate {
    enum { ssize = 100 };

```

```

    T stack[ssize];
    int top;
public:
    StackTemplate() : top(0) {}
    void push(const T& i) {
        require(top < ssize, "Too many push()es");
        stack[top++] = i;
    }
    T pop() {
        require(top > 0, "Too many pop()s");
        return stack[--top];
    }
    int size() { return top; }
};
#endif // STACKTEMPLATE_H ///:~

```

注意，模板会对它包含的对象做一定的假设。例如，**StackTemplate**假设在**push()**函数中有一些对**T**的赋值运算。可以说，模板对于它可以包含的类型“隐含着一个界面”。

表述它的另一种方法是认为模板为C++提供了一种弱类型 (*weak typing*) 机制，C++通常是强类型语言。弱类型不是坚持一个类型是某个可接受的确切类型，而是只要求它想调用的成员函数对于一个特定对象可用就行了。这样，弱类型代码适用于可以接受这些成员函数调用的任何对象，因此更灵活<sup>①</sup>。

这里有一个用于检测模板的修正过的例子：

```

//: C16:StackTemplateTest.cpp
// Test simple stack template
//{L} fibonacci
#include "fibonacci.h"
#include "StackTemplate.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
    StackTemplate<int> is;
    for(int i = 0; i < 20; i++)
        is.push(fibonacci(i));
    for(int k = 0; k < 20; k++)
        cout << is.pop() << endl;
    ifstream in("StackTemplateTest.cpp");
    assure(in, "StackTemplateTest.cpp");
    string line;
    StackTemplate<string> strings;
    while(getline(in, line))
        strings.push(line);
    while(strings.size() > 0)
        cout << strings.pop() << endl;
} ///:~

```

惟一的不同是在实例**is**的创建中。在这个模板参数列表中，我们指明了栈和迭代器应当存

① 在Smalltalk和Python语言中的所有方法都是弱类型，所以这些语言不需要模板机制。实际上，我们得到了无模板的模板。

放的类型。为了显示这个模板的一般性，我们还创建了一个**StackTemplate**来存放**string**。这是通过读入来自源代码文件的代码行来检测的。

### 16.3.3 模板中的常量

模板参数并不局限于类定义的类型，可以使用编译器内置类型。这些参数值在编译期间变成模板的特定示例的常量。我们甚至可以对这些参数使用默认值。下面的例子允许我们在实例化时设置**Array**类的长度，并且还可以提供默认值。

```

//: C16:Array3.cpp
// Built-in types as template arguments
#include "../require.h"
#include <iostream>
using namespace std;

template<class T, int size = 100>
class Array {
    T array[size];
public:
    T& operator[](int index) {
        require(index >= 0 && index < size,
            "Index out of range");
        return array[index];
    }
    int length() const { return size; }
};

class Number {
    float f;
public:
    Number(float ff = 0.0f) : f(ff) {}
    Number& operator=(const Number& n) {
        f = n.f;
        return *this;
    }
    operator float() const { return f; }
    friend ostream&
        operator<<(ostream& os, const Number& x) {
            return os << x.f;
        }
};

template<class T, int size = 20>
class Holder {
    Array<T, size>* np;
public:
    Holder() : np(0) {}
    T& operator[](int i) {
        require(0 <= i && i < size);
        if(!np) np = new Array<T, size>;
        return np->operator[](i);
    }
    int length() const { return size; }
    ~Holder() { delete np; }
};

```



```
};

int main() {
    Holder<Number> h;
    for(int i = 0; i < 20; i++)
        h[i] = i;
    for(int j = 0; j < 20; j++)
        cout << h[j] << endl;
} ///:~
```

如前所述，**Array**是被检查的对象数组，并且防止下标越界。类**Holder**很像**Array**，只是它有一个指向**Array**的指针，而不是指向类型**Array**的嵌入对象。该指针在构造函数中不被初始化，而是推迟到第一次访问时。这称为懒惰初始化 (*lazy initialization*)。如果创造大量的对象，但不访问每一个对象，为了节省存储，可以用懒惰初始化技术。

注意，在这两个模板中，**size**值决不存放在类中，但对它的使用就如同是成员函数中的数据成员。

## 16.4 作为模板的Stash和Stack

贯穿本书反复讨论的**Stash**和**Stack**容器类面临的“所有权”问题，源于我们还不能确切地知道这些容器包含的是什么类型。最近出现的是**Object**的**Stack**容器，这在第15章最后的**OStackTest.cpp**中已经看到了。

如果客户程序员不显式地移去所有指向存放在容器中对象的指针，则容器应当能正确地删除这些指针。这就是说，容器“拥有”不被移走的对象，负责清除它们。问题是这个清除要求关于对象类型的知识，而创造一个一般性的容器类不要求关于对象类型的知识。然而，利用模板，我们可以编写不知道对象类型的代码，并且对于我们希望包含的每种类型，我们可以更容易地实例化这个容器的新版本。个别的已实例化的容器不知道它们保存的对象的类型，但能调用正确的析构函数（假定在典型情况下包含多态性，这时已提供了虚析构函数）。

对于**Stack**，结果很简单，因为所有成员函数都能合理地内联。

```
///: C16:TStack.h
// The Stack as a template
#ifndef TSTACK_H
#define TSTACK_H

template<class T>
class Stack {
    struct Link {
        T* data;
        Link* next;
        Link(T* dat, Link* nxt):
            data(dat), next(nxt) {}
    } * head;
public:
    Stack() : head(0) {}
    ~Stack(){
        while(head)
            delete pop();
    }
    void push(T* dat) {
```



```

    head = new Link(dat, head);
}
T* peek() const {
    return head ? head->data : 0;
}
T* pop(){
    if(head == 0) return 0;
    T* result = head->data;
    Link* oldHead = head;
    head = head->next;
    delete oldHead;
    return result;
}
};
#endif // TSTACK_H ///:~

```

如果将它与第15章最后的例子**OStack.h**相比较，我们可以看到**Stack**实际上相同，除了**Object**已经用**T**替换以外。测试程序也近似相同，除了消除了从**string**和**Object**多重继承的必要性（甚至对于**Object**本身的需要）以外。现在，没有**MyString**类宣布它的销毁，所以增加了一个小的新类来显示**Stack**容器清除它的对象。

```

//: C16:TStackTest.cpp
//{T} TStackTest.cpp
#include "TStack.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

class X {
public:
    virtual ~X() { cout << "~X " << endl; }
};

int main(int argc, char* argv[]) {
    requireArgs(argc, 1); // File name is argument
    ifstream in(argv[1]);
    assure(in, argv[1]);
    Stack<string> textlines;
    string line;
    // Read file and store lines in the Stack:
    while(getline(in, line))
        textlines.push(new string(line));
    // Pop some lines from the stack:
    string* s;
    for(int i = 0; i < 10; i++) {
        if((s = (string*)textlines.pop())==0) break;
        cout << *s << endl;
        delete s;
    } // The destructor deletes the other strings.
    // Show that correct destruction happens:
    Stack<X> xx;
    for(int j = 0; j < 10; j++)
        xx.push(new X);
} ///:~

```



**X**的析构函数是虚的，这里不是因为需要如此，而是因为**xx**稍后能用来存放从**X**派生的对象。

注意，对于**string**和对于**X**创造不同种类的**Stack**是多么容易。由于模板的存在，我们可以得到两方面的好处——即**Stack**类容易使用和正确清除。

### 16.4.1 模板化的指针Stash

重新组织**PStash**代码成为模板并不简单，因为有一些成员函数不应当内联。但是，作为一个模板，这些函数定义仍然存放在头文件中（编译器和连接器处理多定义问题）。代码看上去非常类似于通常的**PStash**，除了增量的大小（由**inflate()**使用）已经被模板化为具有默认值的无类参数以外，所以这个增量的大小能在实例化时修改（注意，这意味着增量大小是固定的，尽管有人会争辩增量大小应当在对象的整个生命期中都是可以改变的）。

```
//: C16:TPStash.h
#ifndef TPSTASH_H
#define TPSTASH_H

template<class T, int incr = 10>
class PStash {
    int quantity; // Number of storage spaces
    int next; // Next empty space
    T** storage;
    void inflate(int increase = incr);
public:
    PStash() : quantity(0), next(0), storage(0) {}
    ~PStash();
    int add(T* element);
    T* operator[](int index) const; // Fetch
    // Remove the reference from this PStash:
    T* remove(int index);
    // Number of elements in Stash:
    int count() const { return next; }
};

template<class T, int incr>
int PStash<T, incr>::add(T* element) {
    if(next >= quantity)
        inflate(incr);
    storage[next++] = element;
    return(next - 1); // Index number
}

// Ownership of remaining pointers:
template<class T, int incr>
PStash<T, incr>::~~PStash() {
    for(int i = 0; i < next; i++) {
        delete storage[i]; // Null pointers OK
        storage[i] = 0; // Just to be safe
    }
    delete []storage;
}

template<class T, int incr>
T* PStash<T, incr>::operator[](int index) const {
```





```

    require(index >= 0,
        "PStash::operator[] index negative");
    if(index >= next)
        return 0; // To indicate the end
    require(storage[index] != 0,
        "PStash::operator[] returned null pointer");
    // Produce pointer to desired element:
    return storage[index];
}

template<class T, int incr>
T* PStash<T, incr>::remove(int index) {
    // operator[] performs validity checks:
    T* v = operator[](index);
    // "Remove" the pointer:
    if(v != 0) storage[index] = 0;
    return v;
}

template<class T, int incr>
void PStash<T, incr>::inflate(int increase) {
    const int psz = sizeof(T*);
    T** st = new T*[quantity + increase];
    memset(st, 0, (quantity + increase) * psz);
    memcpy(st, storage, quantity * psz);
    quantity += increase;
    delete []storage; // Old storage
    storage = st; // Point to new memory
}
#endif // TPSTASH_H ///:~

```

在这里使用的默认增量大小是很小的，以便保证能发生对`inflate()`的调用。我们采用的这种方法可以确保工作正确。

为了测试模板化的**PStash**的控制权，下面的类将报告自身的创建和销毁，并保证被创建的对象都能被销毁。**AutoCounter**只允许它的类型的对象在栈上创建。

```

//: C16:AutoCounter.h
#ifndef AUTOCOUNTER_H
#define AUTOCOUNTER_H
#include "../require.h"
#include <iostream>
#include <set> // Standard C++ Library container
#include <string>

class AutoCounter {
    static int count;
    int id;
    class CleanupCheck {
        std::set<AutoCounter*> trace;
    public:
        void add(AutoCounter* ap) {
            trace.insert(ap);
        }
        void remove(AutoCounter* ap) {
            require(trace.erase(ap) == 1,

```



```

        "Attempt to delete AutoCounter twice");
    }
    ~CleanupCheck() {
        std::cout << "~CleanupCheck()" << std::endl;
        require(trace.size() == 0,
            "All AutoCounter objects not cleaned up");
    }
};
static CleanupCheck verifier;
AutoCounter() : id(count++) {
    verifier.add(this); // Register itself
    std::cout << "created[" << id << "]"
        << std::endl;
}
// Prevent assignment and copy-construction:
AutoCounter(const AutoCounter&);
void operator=(const AutoCounter&);
public:
    // You can only create objects with this:
    static AutoCounter* create() {
        return new AutoCounter();
    }
    ~AutoCounter() {
        std::cout << "destroying[" << id
            << "]" << std::endl;
        verifier.remove(this);
    }
    // Print both objects and pointers:
    friend std::ostream& operator<< (
        std::ostream& os, const AutoCounter& ac) {
        return os << "AutoCounter " << ac.id;
    }
    friend std::ostream& operator<< (
        std::ostream& os, const AutoCounter* ac) {
        return os << "AutoCounter " << ac->id;
    }
};
#endif // AUTOCOUNTER_H ///:~

```

**AutoCounter**类做两件事。第一，它继续对**AutoCounter**的每个实例编号：这个编号的值保存在**id**中，并且使用**static**数据成员**count**来生成这个编号。

第二，更复杂，嵌套类**CleanupCheck**的一个静态实例（称为**verifier**）跟踪被创建和销毁的所有的**AutoCounter**对象，如果程序员没有完全清除它们，它就向程序员报告（也就是假定这里有一个内存泄漏）。这个行为是使用标准C++类库中的**set**类完成的，这是良好设计的模板如何能方便使用的极好例子（在本书的第2卷，我们可以学习C++标准类库中的所有容器）。

**set**类是按照它所包含的类型建立模板的；在这里，它被实例化为包含**AutoCounter**指针的实例。一个**set**只允许每个不同对象的一个实例被添加；在**add()**中，这由**set::insert()**函数完成。如果我们正在试图添加先前已经添加过的内容，**insert()**就用它的返回值通知我们。然而，因为对象地址被添加，所以我们可以依靠C++保证所有对象有惟一的地址。

在**remove()**中，使用**set::erase()**从**set**中移出**AutoCounter**指针。返回值告诉我们这个元素的多少个实例被移出。在这种情况下，我们只希望返回0或1。如果返回值是0，表示这个对

象已经从set中删除，并且这是第二次试图删除它，这是一个程序设计错误，可以通过require()报告这个错误。

CleanupCheck的析构函数最后检查set的长度是否确实是0。如果是0，表示它的所有对象都已经被完全清除。如果不是0，说明有内存泄漏，可以通过require()报告这个错误。

AutoCounter的构造函数和析构函数用verifier对象注册和注销它们自己。注意，构造函数、拷贝构造函数以及赋值运算符都是private的，所以创建对象的惟一方法是用static create()成员函数，这是factory的一个简单例子，它保证所有的对象都在堆上创建，所以verifier对于赋值和拷贝构造不会混淆。

因为所有的成员函数都是内联的，所以使用实现文件的惟一原因是为了包含静态数据成员的定义。

```
//: C16:AutoCounter.cpp {0}
// Definition of static class members
#include "AutoCounter.h"
AutoCounter::CleanupCheck AutoCounter::verifier;
int AutoCounter::count = 0;
///:~
```

利用手边的AutoCounter，我们现在可以测试PStash的功能。下面的例子不仅表明PStash析构函数清除了它现在所拥有的对象，而且还表明AutoCounter类如何检测到还没有被清除的对象。

```
//: C16:TPStashTest.cpp
//{L} AutoCounter
#include "AutoCounter.h"
#include "TPStash.h"
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    PStash<AutoCounter> acStash;
    for(int i = 0; i < 10; i++)
        acStash.add(AutoCounter::create());
    cout << "Removing 5 manually:" << endl;
    for(int j = 0; j < 5; j++)
        delete acStash.remove(j);
    cout << "Remove two without deleting them:"
        << endl;
    // ... to generate the cleanup error message.
    cout << acStash.remove(5) << endl;
    cout << acStash.remove(6) << endl;
    cout << "The destructor cleans up the rest:"
        << endl;
    // Repeat the test from earlier chapters:
    ifstream in("TPStashTest.cpp");
    assure(in, "TPStashTest.cpp");
    PStash<string> stringStash;
    string line;
    while(getline(in, line))
        stringStash.add(new string(line));
    // Print out the strings:
```



```

    for(int u = 0; stringStash[u]; u++)
        cout << "stringStash[" << u << "] = "
            << *stringStash[u] << endl;
} ///:~

```

当从**PStash**中移出**AutoCounter**元素5和元素6时，它们就变成了调用者的责任，但是因为调用者没有清除它们，所以就引起了内存泄漏，它们随后在运行时被**AutoCounter**检测到。

当我们运行这个程序时，会看到错误信息不像希望的那样详细。如果在系统中使用**AutoCounter**中所描述的方案去发现内存泄漏，也许希望打印出关于未被清除对象的更详细的信息。本书的第2卷表明了做这件事情的更好方法。

## 16.5 打开和关闭所有权

让我们回到所有权问题上来。以值包含对象的容器通常无须担心所有权问题，因为它们清晰地拥有它们所包含的对象。但是，如果容器内包含指向对象的指针（这种情况在C++中相当普遍，尤其在多态情况下），而这些指针很可能用于程序的其他地方，那么删除该指针指向的对象会导致在程序的其他地方的指针对已销毁的对象进行引用。为了避免上述情况，在设计和使用容器时必须考虑所有权问题。

许多程序都比这个问题更简单，并且不会遇到所有权问题：一个容器所包含的指针指向仅由这个容器使用的那些对象。在这种情况下，所有权简单而直观：该容器拥有它自己的对象。

处理所有权问题的最好方法是由客户程序员来选择。这常常通过构造函数的一个参数来完成，它默认地指明所有权（简单情况）。另外还有“读取”和“设置”函数用来查看和修正容器的所有权。如果容器内有用于删除对象的函数，容器所有权的状态通常会影响这个删除，所以我们还可以找到在删除函数中控制销毁的选项。我们可以对容器中的每一个成员添加所有权数据，这样每个位置都知道它是否需要被销毁；这是一个引用计数的变体，在这里是容器而不是对象知道所指对象的引用数。

```

//: C16:OwnerStack.h
// Stack with runtime controllable ownership
#ifdef OWNERSTACK_H
#define OWNERSTACK_H

template<class T> class Stack {
    struct Link {
        T* data;
        Link* next;
        Link(T* dat, Link* nxt)
            : data(dat), next(nxt) {}
    }* head;
    bool own;
public:
    Stack(bool own = true) : head(0), own(own) {}
    ~Stack();
    void push(T* dat) {
        head = new Link(dat, head);
    }
    T* peek() const {
        return head ? head->data : 0;
    }
};

```



```

    }
    T* pop();
    bool owns() const { return own; }
    void owns(bool newownership) {
        own = newownership;
    }
    // Auto-type conversion: true if not empty:
    operator bool() const { return head != 0; }
};

```

```

template<class T> T* Stack<T>::pop() {
    if(head == 0) return 0;
    T* result = head->data;
    Link* oldHead = head;
    head = head->next;
    delete oldHead;
    return result;
}

```

```

template<class T> Stack<T>::~~Stack() {
    if(!own) return;
    while(head)
        delete pop();
}
#endif // OWNERSTACK_H ///:~

```

默认行为是让容器去销毁它的对象，但我们可以通过修改构造函数的参数或者使用**owns()**读/写成员函数来改变这个行为。

正如我们可能看到的大多数模板那样，整个实现包含在头文件中。下面是一个检验所有权能力的小测试。

```

//: C16:OwnerStackTest.cpp
//{L} AutoCounter
#include "AutoCounter.h"
#include "OwnerStack.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
    Stack<AutoCounter> ac; // Ownership on
    Stack<AutoCounter> ac2(false); // Turn it off
    AutoCounter* ap;
    for(int i = 0; i < 10; i++) {
        ap = AutoCounter::create();
        ac.push(ap);
        if(i % 2 == 0)
            ac2.push(ap);
    }
    while(ac2)
        cout << ac2.pop() << endl;
    // No destruction necessary since
    // ac "owns" all the objects
} ///:~

```



**ac2**对象不拥有放在它里面的对象，因而**ac**就是对所有权员有责任的“主”容器。在容器生命期内如果希望改变一个容器拥有它的对象，我们可以用**owns()**做这件事。

我们还有可能改变所有权的粒度，使得它以“object-by-object”为基础，但是这将可能会使所有权问题的解决更趋复杂。

## 16.6 以值存放对象

实际上，如果我们没有模板，那么在一个一般的容器内创建对象的一个拷贝是一个复杂的问题。使用模板，事情就相对简单了，只要说我们存放对象而不是指针就行了。

```
//: C16:ValueStack.h
// Holding objects by value in a Stack
#ifndef VALUESTACK_H
#define VALUESTACK_H
#include "../require.h"
template<class T, int ssize = 100>
class Stack {
    // Default constructor performs object
    // initialization for each element in array:
    T stack[ssize];
    int top;
public:
    Stack() : top(0) {}
    // Copy-constructor copies object into array:
    void push(const T& x) {
        require(top < ssize, "Too many push()es");
        stack[top++] = x;
    }
    T peek() const { return stack[top]; }
    // Object still exists when you pop it;
    // it just isn't available anymore:
    T pop() {
        require(top > 0, "Too many pop()s");
        return stack[--top];
    }
};
#endif // VALUESTACK_H ///:~
```

用于被包含对象的拷贝构造函数通过按值传递和返回对象来做大部分工作。在**push()**内，对象在**Stack**数组上的对象存储是用**T::operator=**完成的。为了保证工作，一个称为**SelfCounter**的类将跟踪对象创建和拷贝构造。

```
//: C16:SelfCounter.h
#ifndef SELF_COUNTER_H
#define SELF_COUNTER_H
#include "ValueStack.h"
#include <iostream>

class SelfCounter {
    static int counter;
    int id;
public:
    SelfCounter() : id(counter++) {
```

```

        std::cout << "Created: " << id << std::endl;
    }
    SelfCounter(const SelfCounter& rv) : id(rv.id) {
        std::cout << "Copied: " << id << std::endl;
    }
    SelfCounter operator=(const SelfCounter& rv) {
        std::cout << "Assigned " << rv.id << " to "
            << id << std::endl;
        return *this;
    }
    ~SelfCounter() {
        std::cout << "Destroyed: " << id << std::endl;
    }
    friend std::ostream& operator<< (
        std::ostream& os, const SelfCounter& sc) {
        return os << "SelfCounter: " << sc.id;
    }
};
#endif // SELFCOUNTER_H ///:~

//: C16:SelfCounter.cpp {0}
#include "SelfCounter.h"
int SelfCounter::counter = 0; ///:~

//: C16:ValueStackTest.cpp
//{L} SelfCounter
#include "ValueStack.h"
#include "SelfCounter.h"
#include <iostream>
using namespace std;

int main() {
    Stack<SelfCounter> sc;
    for(int i = 0; i < 10; i++)
        sc.push(SelfCounter());
    // OK to peek(), result is a temporary:
    cout << sc.peek() << endl;
    for(int k = 0; k < 10; k++)
        cout << sc.pop() << endl;
} ///:~

```

当创建一个**Stack**容器时，对于数组中的每个对象调用被包含对象的默认构造函数。最初将看到100个**SelfCounter**对象被创建，但是，这只是这个数组的初始化。这样的代价可能有点昂贵，但是在像这样的简单设计中没有办法。如果允许**Stack**的规模动态增长，让它更一般化，就会出现更复杂的情况，因为在上面显示的实现中会包括：创建一个新的（更大）的数组，拷贝老的数组给新的数组，销毁这个老的数组（事实上，这是标准C++库函数**vector**类所做的事情）。

## 16.7 迭代器简介

迭代器（*iterator*）是一个对象，它在其他对象的容器上遍历，每次选择它们中的一个，不需要提供对这个容器的实现的直接访问。迭代器提供了一种访问元素的标准方法，无论容器是否提供了直接访问元素的方法。迭代器常常与容器类联合使用，而且迭代器在标准C++

容器的设计和使用中是一个基本概念，这方面的知识在本书的第2卷（可从[www.BruceEckel.com](http://www.BruceEckel.com)下载）中有全面的描述。迭代器也是一种设计模式（*design pattern*），这是第2卷中的有一章的主题。

在许多情况下，迭代器是一个“灵巧指针”；并且事实上，我们会注意到：迭代器通常模仿大多数指针的运算。然而，不同的是，迭代器的设计更安全，所以数组越界的可能性更小（或者说，如果有数组越界，就会更早被发现）。

考虑本章的第一个例子，这里增加了一个简单的迭代器。

```
//: C16:IterIntStack.cpp
// Simple integer stack with iterators
//{L} fibonacci
#include "fibonacci.h"
#include "../require.h"
#include <iostream>
using namespace std;

class IntStack {
    enum { ssize = 100 };
    int stack[ssize];
    int top;
public:
    IntStack() : top(0) {}
    void push(int i) {
        require(top < ssize, "Too many push()es");
        stack[top++] = i;
    }
    int pop() {
        require(top > 0, "Too many pop()s");
        return stack[--top];
    }
    friend class IntStackIter;
};

// An iterator is like a "smart" pointer:
class IntStackIter {
    IntStack& s;
    int index;
public:
    IntStackIter(IntStack& is) : s(is), index(0) {}
    int operator++() { // Prefix
        require(index < s.top,
            "iterator moved out of range");
        return s.stack[++index];
    }
    int operator++(int) { // Postfix
        require(index < s.top,
            "iterator moved out of range");
        return s.stack[index++];
    }
};

int main() {
    IntStack is;
    for(int i = 0; i < 20; i++)
```





```

        is.push(fibonacci(i));
    // Traverse with an iterator:
    IntStackIter it(is);
    for(int j = 0; j < 20; j++)
        cout << it++ << endl;
} ///:~

```

创建**IntStackIter**，以便只与**IntStack**一起工作。注意，**IntStackIter**是**IntStack**的友元，这就允许访问**IntStack**的所有私有成员。

像指针一样，**IntStackIter**的工作是遍历**IntStack**，并提取值。在这个简单的例子中，**IntStackIter**只能向前移动（用**operator++**的前缀和后缀形式）。然而，迭代器定义没有限制，而只是有与它一起工作的容器的约束限制。在与迭代器相联系的容器中，迭代器可以用任何方式移动，并且可以通过它修改被包含的值。

习惯上，用构造函数来创建迭代器，并把它与一个容器对象联系，并且在它的生命期中，不把它与不同的容器联系。（迭代器通常是小的和廉价的，所以可以很容易地再做一个。）

使用迭代器，我们可以扫描栈的元素而不用弹出它们，这就像指针遍历数组的元素一样。然而，迭代器知道栈的下层结构，并知道如何遍历栈的元素，所以即便我们正在以“向前移动指针”的方式遍历栈的元素，我们也应该使用迭代器。下面将介绍更多的内容。迭代器的关键是：从一个容器元素移动到下一个元素的复杂过程被抽象为就像一个指针一样。目标是使程序中的每个迭代器都有相同的接口，使得使用这个迭代器的任何代码都不用关心它指向什么，只需要知道它能用同样的方法重新配置所有的迭代器，所以这个迭代器所指向的容器并不重要。用这种方法，我们可以编写更一般性的代码。标准C++库的所有容器和算法都基于迭代器的这一原则。

为了让事情更一般化，最好能说“每一个容器有一个相关的名为**iterator**的类”，但是这引起典型的名字问题。解决的办法是为每个容器增加一个嵌套的**iterator**类（注意，在这种情况下，“**iterator**”以小写字母开始，使得它与标准C++库的风格一致）。下面是**IterIntStack.cpp**，带有一个嵌套的**iterator**。

```

//: C16:NestedIterator.cpp
// Nesting an iterator inside the container
//{L} fibonacci
#include "fibonacci.h"
#include "../require.h"
#include <iostream>
#include <string>
using namespace std;

class IntStack {
    enum { ssize = 100 };
    int stack[ssize];
    int top;
public:
    IntStack() : top(0) {}
    void push(int i) {
        require(top < ssize, "Too many push()es");
        stack[top++] = i;
    }
    int pop() {

```



```

        require(top > 0, "Too many pop()s");
        return stack[--top];
    }
    class iterator;
    friend class iterator;
    class iterator {
        IntStack& s;
        int index;
    public:
        iterator(IntStack& is) : s(is), index(0) {}
        // To create the "end sentinel" iterator:
        iterator(IntStack& is, bool)
            : s(is), index(s.top) {}
        int current() const { return s.stack[index]; }
        int operator++() { // Prefix
            require(index < s.top,
                "iterator moved out of range");
            return s.stack[++index];
        }
        int operator++(int) { // Postfix
            require(index < s.top,
                "iterator moved out of range");
            return s.stack[index++];
        }
        // Jump an iterator forward
        iterator& operator+=(int amount) {
            require(index + amount < s.top,
                "IntStack::iterator::operator+=(int) "
                "tried to move out of bounds");
            index += amount;
            return *this;
        }
        // To see if you're at the end:
        bool operator==(const iterator& rv) const {
            return index == rv.index;
        }
        bool operator!=(const iterator& rv) const {
            return index != rv.index;
        }
        friend ostream&
        operator<<(ostream& os, const iterator& it) {
            return os << it.current();
        }
    };
    iterator begin() { return iterator(*this); }
    // Create the "end sentinel":
    iterator end() { return iterator(*this, true); }
};

int main() {
    IntStack is;
    for(int i = 0; i < 20; i++)
        is.push(fibonacci(i));
    cout << "Traverse the whole IntStack\n";
    IntStack::iterator it = is.begin();
    while(it != is.end())

```

```

        cout << it++ << endl;
    cout << "Traverse a portion of the IntStack\n";
    IntStack::iterator
        start = is.begin(), end = is.begin();
    start += 5, end += 15;
    cout << "start = " << start << endl;
    cout << "end = " << end << endl;
    while(start != end)
        cout << start++ << endl;
} ///:~

```

当创建一个嵌套**friend**类的时候，我们必须经过首先声明这个类的名称，然后声明它是友元，最后定义这个类的过程。否则，编译器将会产生混淆。

我们在迭代器中增加了一些新的手法。**current()**成员函数产生容器中的由迭代器当前选择的元素。我们可以用**operator+=**使迭代器向前“跳跃”任意个元素。而且，我们还会看到两个重载运算符：**==**和**!=**，它们将比较两个迭代器。它们能比较任意两个**IntStack::iterator**，但是它们的最初意图是测试这个迭代器是否已经到达了序列的终点，采用“实际的”标准C++库迭代器所用的相同方法。其思想是，两个迭代器定义了一个范围，第一个迭代器指向第一个元素，第二个迭代器指向最后一个元素后面的位置。如果希望遍历由这两个迭代器所定义的范围，可以写为如下形式：

```

while(start != end)
    cout << start++ << endl;

```

这里，**start**和**end**是在这个范围内的两个迭代器。注意，**end**迭代器并不反向引用，只是告诉我们已经到了这个范围的终点，我们称之为“终止哨兵”(*end sentinel*)。因而它代表“终点后面的一个”。

大多数情况下我们希望在容器中遍历整个序列，所以这个容器需要某种方法产生表示这个序列的开始和终止哨兵的迭代器。在此，就像在标准C++库中一样，这些迭代器由容器的成员函数**begin()**和**end()**产生。**begin()**使用第一个迭代器构造函数，它默认指向这个容器的开始（这就是压入这个栈的第一个元素）。然而，第二个构造函数，由**end()**使用，对于创建终止哨兵迭代器是必需的。“在终点”的意思是指向这个栈的顶部，因为**top**允许指向下一个可用的但是尚未使用的位置。这个迭代器构造函数采用第二个类型为**bool**的参数，它是哑元，以区别两个构造函数。

在**main()**中再次使用斐波纳契数来填充**IntStack**，用迭代器遍历整个**IntStack**并且还遍历序列的一个小范围。

当然，下一步是通过对它所包含的类型模板化来让代码一般化，所以不是强迫仅能存放**int**，而是可以存放任何类型。

```

//: C16:IterStackTemplate.h
// Simple stack template with nested iterator
#ifdef ITERSTACKTEMPLATE_H
#define ITERSTACKTEMPLATE_H
#include "../require.h"
#include <iostream>

template<class T, int ssize = 100>
class StackTemplate {

```

```

T stack[ssize];
int top;
public:
    StackTemplate() : top(0) {}
    void push(const T& i) {
        require(top < ssize, "Too many push()es");
        stack[top++] = i;
    }
    T pop() {
        require(top > 0, "Too many pop()s");
        return stack[--top];
    }
    class iterator; // Declaration required
    friend class iterator; // Make it a friend
    class iterator { // Now define it
        StackTemplate& s;
        int index;
    public:
        iterator(StackTemplate& st): s(st), index(0) {}
        // To create the "end sentinel" iterator:
        iterator(StackTemplate& st, bool)
            : s(st), index(s.top) {}
        T operator*() const { return s.stack[index]; }
        T operator++() { // Prefix form
            require(index < s.top,
                "iterator moved out of range");
            return s.stack[++index];
        }
        T operator++(int) { // Postfix form
            require(index < s.top,
                "iterator moved out of range");
            return s.stack[index++];
        }
        // Jump an iterator forward
        iterator& operator+=(int amount) {
            require(index + amount < s.top,
                " StackTemplate::iterator::operator+=( ) "
                "tried to move out of bounds");
            index += amount;
            return *this;
        }
        // To see if you're at the end:
        bool operator==(const iterator& rv) const {
            return index == rv.index;
        }
        bool operator!=(const iterator& rv) const {
            return index != rv.index;
        }
        friend std::ostream& operator<<(
            std::ostream& os, const iterator& it) {
            return os << *it;
        }
    };
    iterator begin() { return iterator(*this); }
    // Create the "end sentinel":
    iterator end() { return iterator(*this, true); }

```



```
};
#endif // ITERSTACKTEMPLATE_H ///:~
```

可以看到，从正规类到模板的转换是适度透明的。首先创建和调试一个普通类，然后让它成为模板，一般认为这种方法比一开始就创建模板更容易。

注意，不是只写：

```
friend iterator; // Make it a friend
```

这段代码是：

```
friend class iterator; // Make it a friend
```

这是重要的，因为名字“iterator”已经在一个范围内，来自一个被包含的文件。

不是用**current()**成员函数，而是**iterator**有一个**operator\***，用来选择当前的元素，这使**iterator**看上去更像一个指针，这是一个普通的习惯。

下面是一个用来测试模板的修改过的例子：

```
//: C16:IterStackTemplateTest.cpp
//{L} fibonacci
#include "fibonacci.h"
#include "IterStackTemplate.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
    StackTemplate<int> is;
    for(int i = 0; i < 20; i++)
        is.push(fibonacci(i));
    // Traverse with an iterator:
    cout << "Traverse the whole StackTemplate\n";
    StackTemplate<int>::iterator it = is.begin();
    while(it != is.end())
        cout << *it++ << endl;
    cout << "Traverse a portion\n";
    StackTemplate<int>::iterator
        start = is.begin(), end = is.begin();
    start += 5, end += 15;
    cout << "start = " << start << endl;
    cout << "end = " << end << endl;
    while(start != end)
        cout << *start++ << endl;
    ifstream in("IterStackTemplateTest.cpp");
    assure(in, "IterStackTemplateTest.cpp");
    string line;
    StackTemplate<string> strings;
    while(getline(in, line))
        strings.push(line);
    StackTemplate<string>::iterator
        sb = strings.begin(), se = strings.end();
    while(sb != se)
        cout << *sb++ << endl;
} ///:~
```



迭代器的第一个应用只移动它从开始到最后（可以看到终止哨兵工作正常）。在第二个应用中，我们可以看到迭代器如何允许我们容易地指定元素的范围（在标准C++库中，容器和迭代器随处使用范围的概念）。重载`operator+=`移动`start`和`end`迭代器到`is`中元素范围的中间位置，打印出这些元素。注意，在输出中，终止哨兵不在范围内，这样，它可以是范围终点后面的一个，可以让程序员知道他已经越过了终点，但是，不反向引用终止哨兵，否则就相当于反向引用空指针。（在`StackTemplate::iterator`中我已经做了防护，但是在标准C++库中的容器和迭代器中，出于效率的原因，没有这样的代码，所以必须注意。）

最后，为了验证`StackTemplate`与类对象一起工作，采用一个`string`的实例，它用源代码文件中的行填充这些字符串，然后打印出它们。

### 16.7.1 带有迭代器的栈

重复具有动态长度`Stack`类的过程，这是贯穿本书的例子。这里`Stack`类带有一个嵌套的迭代器。

```
//: C16:TStack2.h
// Templated Stack with nested iterator
#ifndef TSTACK2_H
#define TSTACK2_H

template<class T> class Stack {
    struct Link {
        T* data;
        Link* next;
        Link(T* dat, Link* nxt)
            : data(dat), next(nxt) {}
    }* head;
public:
    Stack() : head(0) {}
    ~Stack();
    void push(T* dat) {
        head = new Link(dat, head);
    }
    T* peek() const {
        return head ? head->data : 0;
    }
    T* pop();
    // Nested iterator class:
    class iterator; // Declaration required
    friend class iterator; // Make it a friend
    class iterator { // Now define it
        Stack::Link* p;
    public:
        iterator(const Stack<T>& tl) : p(tl.head) {}
        // Copy-constructor:
        iterator(const iterator& tl) : p(tl.p) {}
        // The end sentinel iterator:
        iterator() : p(0) {}
        // operator++ returns boolean indicating end:
        bool operator++() {
            if(p->next)
                p = p->next;
```



```

        else p = 0; // Indicates end of list
        return bool(p);
    }
    bool operator++(int) { return operator++(); }
    T* current() const {
        if(!p) return 0;
        return p->data;
    }
    // Pointer dereference operator:
    T* operator->() const {
        require(p != 0,
            "PStack::iterator::operator->returns 0");
        return current();
    }
    T* operator*() const { return current(); }
    // bool conversion for conditional test:
    operator bool() const { return bool(p); }
    // Comparison to test for end:
    bool operator==(const iterator&) const {
        return p == 0;
    }
    bool operator!=(const iterator&) const {
        return p != 0;
    }
};
iterator begin() const {
    return iterator(*this);
}
iterator end() const { return iterator(); }
};

template<class T> Stack<T>::~~Stack() {
    while(head)
        delete pop();
}

template<class T> T* Stack<T>::pop() {
    if(head == 0) return 0;
    T* result = head->data;
    Link* oldHead = head;
    head = head->next;
    delete oldHead;
    return result;
}
#endif // TSTACK2_H ///:~

```

我们已经注意到，这个类已经被修改以支持所有权，它能工作是因为这个类知道确切类型（或者至少知道基本类型，这是基于使用虚构造函数的假设而工作的）。对于容器的默认是销毁它的对象，但是我们要负责处理我们`pop()`的任何指针。

迭代器是简单的，体积非常小，即单个指针的大小。当创建一个迭代器时，它被初始化为指向链表的头，只能沿着链表向前递增。如果希望指向起点之后，就创建一个新迭代器，如果希望记住表中的一点，就从已存在的迭代器中创建一个新迭代器，指向这一点（使用迭代器的拷贝构造函数）。

为了对由迭代器指向的对象调用函数，我们可以使用`current()`函数、`operator*`和指针反向引用`operator->`（迭代器中的一个共同点）。后者的实现看上去与`current()`一样，因为它返回一个指向当前对象的指针，但是实际上不同，因为这个指针反向引用运算符完成反向引用的外层（参见第12章）。

`iterator`类遵循前面例子中的形式。`class iterator`嵌套在容器类中，它包含构造函数，可以创建指向容器中一个元素的一个迭代器和一个“终止哨兵”迭代器，并且容器类有用来产生这些迭代器的`begin()`和`end()`方法。（当我们对标准C++库的学习更加深入之后，就会看到：在这里用的名字`iterator`、`begin()`和`end()`已经明确地被推举为标准容器类。在本章的最后将会看到，使用这些容器类就像使用标准C++库容器类一样。）

全部实现都包含在头文件中，所以这里没有单独的`cpp`文件。下面是用来检验迭代器的小测试：

```
//: C16:TStack2Test.cpp
#include "TStack2.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
    ifstream file("TStack2Test.cpp");
    assure(file, "TStack2Test.cpp");
    Stack<string> textlines;
    // Read file and store lines in the Stack:
    string line;
    while(getline(file, line))
        textlines.push(new string(line));
    int i = 0;
    // Use iterator to print lines from the list:
    Stack<string>::iterator it = textlines.begin();
    Stack<string>::iterator* it2 = 0;
    while(it != textlines.end()) {
        cout << it->c_str() << endl;
        it++;
        if(++i == 10) // Remember 10th line
            it2 = new Stack<string>::iterator(it);
    }
    cout << (*it2)->c_str() << endl;
    delete it2;
} ///:~
```

`Stack`是一个示例，用来存放`string`对象，并用来自一个文件的行填充。然后创建一个迭代器，用于遍历这个序列。通过从第一个迭代器拷贝构造第二个迭代器，来记住第10行，然后打印这一行，并且销毁动态创建的迭代器。这里，使用动态对象创建来控制该对象的生命期。

### 16.7.2 带有迭代器的PStash

对于大多数容器类，有迭代器是有意义的。这里对`PStash`类添加一个迭代器：



```

//: C16:TPStash2.h
// Templated PStash with nested iterator
#ifndef TPSTASH2_H
#define TPSTASH2_H
#include "../require.h"
#include <cstdlib>

template<class T, int incr = 20>
class PStash {
    int quantity;
    int next;
    T** storage;
    void inflate(int increase = incr);
public:
    PStash() : quantity(0), storage(0), next(0) {}
    ~PStash();
    int add(T* element);
    T* operator[](int index) const;
    T* remove(int index);
    int count() const { return next; }
    // Nested iterator class:
    class iterator; // Declaration required
    friend class iterator; // Make it a friend
    class iterator { // Now define it
        PStash& ps;
        int index;
    public:
        iterator(PStash& pStash)
            : ps(pStash), index(0) {}
        // To create the end sentinel:
        iterator(PStash& pStash, bool)
            : ps(pStash), index(ps.next) {}
        // Copy-constructor:
        iterator(const iterator& rv)
            : ps(rv.ps), index(rv.index) {}
        iterator& operator=(const iterator& rv) {
            ps = rv.ps;
            index = rv.index;
            return *this;
        }
        iterator& operator++() {
            require(++index <= ps.next,
                "PStash::iterator::operator++ "
                "moves index out of bounds");
            return *this;
        }
        iterator& operator++(int) {
            return operator++();
        }
        iterator& operator--() {
            require(--index >= 0,
                "PStash::iterator::operator-- "
                "moves index out of bounds");
            return *this;
        }
        iterator& operator--(int) {

```



```

        return operator--();
    }
    // Jump iterator forward or backward:
    iterator& operator+=(int amount) {
        require(index + amount < ps.next &&
            index + amount >= 0,
            "PStash::iterator::operator+= "
            "attempt to index out of bounds");
        index += amount;
        return *this;
    }
    iterator& operator-=(int amount) {
        require(index - amount < ps.next &&
            index - amount >= 0,
            "PStash::iterator::operator-= "
            "attempt to index out of bounds");
        index -= amount;
        return *this;
    }
    // Create a new iterator that's moved forward
    iterator operator+(int amount) const {
        iterator ret(*this);
        ret += amount; // op+= does bounds check
        return ret;
    }
    T* current() const {
        return ps.storage[index];
    }
    T* operator*() const { return current(); }
    T* operator->() const {
        require(ps.storage[index] != 0,
            "PStash::iterator::operator->returns 0");
        return current();
    }
    // Remove the current element:
    T* remove() {
        return ps.remove(index);
    }
    // Comparison tests for end:
    bool operator==(const iterator& rv) const {
        return index == rv.index;
    }
    bool operator!=(const iterator& rv) const {
        return index != rv.index;
    }
};

iterator begin() { return iterator(*this); }
iterator end() { return iterator(*this, true); }
};

// Destruction of contained objects:
template<class T, int incr>
PStash<T, incr>::~~PStash() {
    for(int i = 0; i < next; i++) {
        delete storage[i]; // Null pointers OK
        storage[i] = 0; // Just to be safe
    }
}

```



```

    }
    delete []storage;
}

template<class T, int incr>
int PStash<T, incr>::add(T* element) {
    if(next >= quantity)
        inflate();
    storage[next++] = element;
    return(next - 1); // Index number
}

template<class T, int incr> inline
T* PStash<T, incr>::operator[](int index) const {
    require(index >= 0,
        "PStash::operator[] index negative");
    if(index >= next)
        return 0; // To indicate the end
    require(storage[index] != 0,
        "PStash::operator[] returned null pointer");
    return storage[index];
}

template<class T, int incr>
T* PStash<T, incr>::remove(int index) {
    // operator[] performs validity checks:
    T* v = operator[](index);
    // "Remove" the pointer:
    storage[index] = 0;
    return v;
}

template<class T, int incr>
void PStash<T, incr>::inflate(int increase) {
    const int tsz = sizeof(T*);
    T** st = new T*[quantity + increase];
    memset(st, 0, (quantity + increase) * tsz);
    memcpy(st, storage, quantity * tsz);
    quantity += increase;
    delete []storage; // Old storage
    storage = st; // Point to new memory
}

#endif // TPSTASH2_H ///:~

```

这个文件的大部分是先前**PStash**和嵌套**iterator**直接翻译成的模板。然而，这时运算符返回对当前迭代器的引用，这是更典型和更灵活的方法。

析构函数对于所有被包含的指针调用**delete**，并且因为类型由模型获取，所以将发生适当的销毁。应当知道，如果容器存放指向基类类型的指针，那么这个类型应当有虚析构函数，以保证正确地清除派生对象，当将这些派生对象放进该容器时，它们的地址已经发生了向上类型转换。

**PStash::iterator**遵循迭代器在其生命期内只结合单个容器对象这一模式。另外，拷贝构造函数允许让新迭代器指向已存在迭代器指向的同一位置，这就像在容器内夹了书签。**operator++**和**operator--**成员函数允许移动迭代器一些距离，但与容器的边界有关。重载的增

加和减小运算符移动迭代器一个位置。**operator+**生成新迭代器，它向前移动加数个位置。像前面的例子一样，指针反向引用运算符被用于在迭代器涉及的元素上进行运算，**remove()**通过调用容器的**remove()**来销毁当前对象。

就像前面的同样代码（按照标准C++库容器方式），被用来创建终止哨兵：第二构造函数、容器的**end()**成员函数和用于比较的**operator==**与**operator!=**。

下面的例子创建和测试两个不同种类的**Stash**对象：一个成为名为**Int**的新类，它宣布它的构造和析构；另一个存放标准库**string**类的对象。

```
//: C16:TPStash2Test.cpp
#include "TPStash2.h"
#include "../require.h"
#include <iostream>
#include <vector>
#include <string>
using namespace std;
class Int {
    int i;
public:
    Int(int ii = 0) : i(ii) {
        cout << ">" << i << ' ';
    }
    ~Int() { cout << "~" << i << ' '; }
    operator int() const { return i; }
    friend ostream&
        operator<<(ostream& os, const Int& x) {
            return os << "Int: " << x.i;
        }
    friend ostream&
        operator<<(ostream& os, const Int* x) {
            return os << "Int: " << x->i;
        }
};

int main() {
    { // To force destructor call
        PStash<Int> ints;
        for(int i = 0; i < 30; i++)
            ints.add(new Int(i));
        cout << endl;
        PStash<Int>::iterator it = ints.begin();
        it += 5;
        PStash<Int>::iterator it2 = it + 10;
        for(; it != it2; it++)
            delete it.remove(); // Default removal
        cout << endl;
        for(it = ints.begin(); it != ints.end(); it++)
            if(*it) // Remove() causes "holes"
                cout << *it << endl;
    } // "ints" destructor called here
    cout << "\n-----\n";
    ifstream in("TPStash2Test.cpp");
    assure(in, "TPStash2Test.cpp");
    // Instantiate for String:
```



```

PStash<string> strings;
string line;
while(getline(in, line))
    strings.add(new string(line));
PStash<string>::iterator sit = strings.begin();
for(; sit != strings.end(); sit++)
    cout << **sit << endl;
sit = strings.begin();
int n = 26;
sit += n;
for(; sit != strings.end(); sit++)
    cout << n++ << ": " << **sit << endl;
} ///:~

```

为了方便，**Int**有一个相关的**ostream operator<<**，用于**Int&**和**Int\***。

在**main()**中的第一个代码段用花括号括起来，用来强迫**PStash<Int>**的销毁，并因而由析构函数自动清除。用手工移走和删除元素的范围，以表明**PStash**清除了剩余的元素。

对于**PStash**的这两个实例，创建一个迭代器并用于遍历容器。注意由使用这些构造函数所产生的简洁性，我们不会遭受使用数组的实现细节的困扰。只要告诉容器和迭代器对象做什么，而无需告诉它们如何做。这就使得这个解决方法容易概念化、建立和修改。

## 16.8 为什么使用迭代器

到目前为止，我们已经看到了迭代器的机制，但是要理解为什么它们如此重要还需要采用更复杂的例子。

在一个真实的面向对象程序中，经常可以看到多态性、动态对象创建和容器在一起使用。容器和动态对象创建解决了不知道我们需要多少对象以及对象是什么类型的这样的问题。如果配置一个容器存放指向基类对象的指针，每次放置一个派生类指针进入容器时，就发生向上类型转换。正如本书第1卷中最后的代码，这个例子还将我们至今已经学习过的各个不同方面放在一起，如果我们能理解这个例子，我们就为学习第2卷做好了准备。

假设我们正在创建一个程序，这个程序允许用户编辑和产生不同种类的图画。每个图画都是一个包含一组**Shape**对象的对象。

```

//: C16:Shape.h
#ifndef SHAPE_H
#define SHAPE_H
#include <iostream>
#include <string>

class Shape {
public:
    virtual void draw() = 0;
    virtual void erase() = 0;
    virtual ~Shape() {}
};

class Circle : public Shape {
public:
    Circle() {}
    ~Circle() { std::cout << "Circle::~~Circle\n"; }
}

```



```

    void draw() { std::cout << "Circle::draw\n"; }
    void erase() { std::cout << "Circle::erase\n"; }
};

class Square : public Shape {
public:
    Square() {}
    ~Square() { std::cout << "Square::~~Square\n"; }
    void draw() { std::cout << "Square::draw\n"; }
    void erase() { std::cout << "Square::erase\n"; }
};

class Line : public Shape {
public:
    Line() {}
    ~Line() { std::cout << "Line::~~Line\n"; }
    void draw() { std::cout << "Line::draw\n"; }
    void erase() { std::cout << "Line::erase\n"; }
};
#endif // SHAPE_H ///:~

```

这段代码使用了基类中虚函数的典型结构，这次虚函数在派生类中被重新定义。注意，**Shape**类包含一个虚析构函数，应当将有些东西自动添加到具有虚函数的任何类中。如果一个容器存放指向**Shape**对象的指针或引用，则当对这些对象调用这个虚析构函数时，所有的相关数据都将被正确地清除。

在下面例子中，每一个不同类型的图画都使用了不同种类的模板化容器类：已经在本章定义的**PStash**和**Stack**，以及来自标准C++库的**vector**类。容器的“使用”是极其简单的，并且通常情况下，继承可能不是最好的方法（组合可能更有意义），但是，在这种情况下，继承是一个简单的方法，并没有从这个例子中去掉。

```

//: C16:Drawing.cpp
#include <vector> // Uses Standard vector too!
#include "TPStash2.h"
#include "TStack2.h"
#include "Shape.h"
using namespace std;

// A Drawing is primarily a container of Shapes:
class Drawing : public PStash<Shape> {
public:
    ~Drawing() { cout << "~Drawing" << endl; }
};

// A Plan is a different container of Shapes:
class Plan : public Stack<Shape> {
public:
    ~Plan() { cout << "~Plan" << endl; }
};

// A Schematic is a different container of Shapes:
class Schematic : public vector<Shape*> {
public:
    ~Schematic() { cout << "~Schematic" << endl; }
};

```



```

// A function template:
template<class Iter>
void drawAll(Iter start, Iter end) {
    while(start != end) {
        (*start)->draw();
        start++;
    }
}

int main() {
    // Each type of container has
    // a different interface:
    Drawing d;
    d.add(new Circle);
    d.add(new Square);
    d.add(new Line);
    Plan p;
    p.push(new Line);
    p.push(new Square);
    p.push(new Circle);
    Schematic s;
    s.push_back(new Square);
    s.push_back(new Circle);
    s.push_back(new Line);
    Shape* sarray[] = {
        new Circle, new Square, new Line
    };
    // The iterators and the template function
    // allow them to be treated generically:
    cout << "Drawing d:" << endl;
    drawAll(d.begin(), d.end());
    cout << "Plan p:" << endl;
    drawAll(p.begin(), p.end());
    cout << "Schematic s:" << endl;
    drawAll(s.begin(), s.end());
    cout << "Array sarray:" << endl;
    // Even works with array pointers:
    drawAll(sarray,
        sarray + sizeof(sarray)/sizeof(*sarray));
    cout << "End of main" << endl;
} ///:~

```

不同类型的容器都存放指向**Shape**的指针和指向**Shape**派生类的向上类型转换对象的指针。然而，因为多态性，当调用虚函数时，仍然出现正确的行为。

注意，**Shape\***的数组**sarray**也可以被看做一个容器。

### 16.8.1 函数模板

在**drawAll()**中，我们已经看到了一些新东西。但是，到本章为止，我们仅仅使用了类模板，它们实例化基于一个或多个类型参数的新表。然而，我们可以同样容易地创建函数模板，它们创建基于类型参数的新函数。创建函数模板的理由与使用类模板的理由相同：我们试图创建一般性的代码，我们可以通过延迟规定一个或多个类型的方法来创建这样的代码。我们只想写明这些类型参数支持特定运算，并不确切地说明它们是什么类型。

函数模板 `drawAll()` 可以看做是一个算法（在标准C++库中大部分函数模板被称为算法）。它只是给出描述元素的一个区域的迭代器，说明如何做某件事情，只要这些迭代器能被反向引用、增加和比较。在本章中，我们已经开发出这种迭代器，但这也不是巧合，这种迭代器由标准C++库中的容器生成，在这个例子中由使用 `vector` 证实。

我们还希望 `drawAll()` 是一个泛型算法 (*generic algorithm*)，所以容器可以是任意类型的，我们没有必要为每个不同类型的容器编写这个算法的新版本。在此，函数模板是基本的，因为它们能自动地为每个不同类型的容器产生特殊代码。但是，如果没有由迭代器提供的另外的间接性，这种泛型 (*genericness*) 就没有可能。这就是迭代器为什么如此重要的原因；它们允许用户编写涉及容器的通用代码，而用户并不知道容器的下层结构。（注意，在C++中，为了正确工作，迭代器和泛型算法都需要函数模板。）

在 `main()` 中可以看到这点的证明，因为 `drawAll()` 的工作不随着容器类型的不同而改变。更有趣的是，`drawAll()` 对于指向数组 `sarray` 的开始和结尾的指针也能工作。这种将数组作为容器处理的能力是标准C++库设计的一部分，它们的算法很像 `drawAll()`。

因为容器类模板很少关系到普通类所具有的继承和向上类型转换，所以不会在容器类中看到虚函数。容器的重用是用模板，而不是用继承实现的。

## 16.9 小结

容器类是面向对象程序设计的一个基本部分。它们是简化和隐藏实施细节、提高开发效率的另一种方法。另外，它们通过替换C语言中发现的原始数组和相对粗糙的数据结构技术从而大大地提高了程序的灵活性和安全性。

因为客户程序员需要容器，所以容器的便于使用是它的基本特征。这样，模板就被引入。使用模板语法，对源代码进行的重用（相反的是，由继承和组合提供的对对象代码进行的重用）对初学者来说变得十分平常。实际上，使用模板实施代码重用比使用继承和组合实施代码重用容易得多。

虽然在本书中我们已经学习了创建容器和迭代器类的相关知识，但实际上，更有用的是学习了在标准C++库中的容器和迭代器，因为可以期望在每个编译器中使用它们。正如我们将会在本书的第2卷（可从 [www.BruceEckel.com](http://www.BruceEckel.com) 处下载）中看到的，标准C++库中的容器和算法实际上总能满足我们的需要，因此不需要自己创建新的容器类。

本章已经涉及与容器类设计有关的问题，但我们可能希望学习更多的内容。一个更加复杂的容器类库可以覆盖所有的其他问题，包括多线程、持久存储和无用单元收集。

### 16.10 练习

部分练习题的答案可以在本书的电子文档 “*Annotated Solution Guide for Thinking in C++*” 中找到，只需支付很少的费用就可以从 <http://www.BruceEckel.com> 得到这个电子文档。

16-1 实现本章中 `OShape` 图的继承层次。

16-2 修改第15章练习1的结果，以便使用 `TStack2.h` 中的 `Stack` 和 `iterator` 替代一个 `Shape` 指针的数组。增加针对类层次的析构函数，使得我们可以观察到：在 `Stack` 超出范围时 `Shape` 对象的销毁。

16-3 修改 `TPStash.h`，使得由 `inflate()` 使用的增量值能在特定容器对象的生命期内改变。



- 16-4 修改**TPStash.h**，使得由**inflate()**使用的增量值能自动地调整自身大小，以减少它需要被调用的次数。例如，每次调用它，它都能为下一次调用而加倍这个增量值。通过报告是否**inflate()**被调用来证明这个功能，并且在**main()**中编写测试代码。
- 16-5 对于**fibonacci()**函数产生的值的类型，模板化**fibonacci()**函数（使得它能产生**long**、**float**等类型的值，而不只产生**int**型值）。
- 16-6 使用标准C++库**vector**作为下层实现，创建一个**Set**模板类，对于放入这个模板类中的每一种对象，它只接受一个。创建一个嵌套**iterator**类，它支持本章中的“终止哨兵”思想。在**main()**中编写测试代码，并随后代替标准C++库的**Set**模板以验证其行为是正确的。
- 16-7 修改**AutoCounter.h**使得它能在任何类中被用作成员对象，我们希望能跟踪它的创建和销毁。增加一个**string**成员，用来保存这个类的名称。在我们自己的一个类中测试这个工具。
- 16-8 创建**OwnerStack.h**的一个版本，它使用标准C++库**vector**作为它的下层实现。为此，我们可能需要查寻**vector**的一些成员函数（或只考虑**<vector>**头文件）。
- 16-9 修改**ValueStack.h**，使得当我们**push()**更多的对象并且超出空间时它能自动地扩展。改动**ValueStackTest.cpp**以测试新的功能性。
- 16-10 重复练习9，但使用标准C++库**vector**作为**ValueStack**的内部实现。注意，这种做法容易多了。
- 16-11 修改**ValueStackTest.cpp**，使得它在**main()**中使用标准C++库**vector**而不使用**Stack**。注意运行时的行为：当**vector**创建时它自动创建一系列默认对象吗？
- 16-12 修改**TStack2.h**，使得它使用标准C++库**vector**作为它的下层实现。确信不要改变接口就能让**TStack2Test.cpp**照常工作。
- 16-13 使用标准C++库**Stack**而不使用**vector**重复练习12（可能需要查寻关于**stack**的信息，或者搜索**<stack>**头文件）。
- 16-14 修改**TPStash2.h**，使得它使用标准C++库**vector**作为它的下层实现。确信不要改变接口就能让**TPStash2Test.cpp**照常工作。
- 16-15 在**IterIntStack.cpp**中，修改**IntStackIter**，给它一个“终止哨兵”构造函数，添加**operator ==**和**operator !=**。在**main()**中，使用一个迭代器遍历这个容器的元素，直到我们到达“终止哨兵”。
- 16-16 使用**TStack2.h**、**TPStash2.h**和**Shape.h**，为**Shape\***实例化**Stack**和**PStash**容器，对它们填充向上类型转换的**Shape**指针，然后用迭代器遍历每个容器，并为每个对象调用**draw()**。
- 16-17 模板化**TPStash2Test.cpp**中的**Int**类，使得它存放任意类型的对象（可以改变这个类的名称，使之更确切）。
- 16-18 模板化来自第12章的**IostreamOperatorOverloading.cpp**中的**IntArray**类，模型化它包含的对象的类型和内部数组的长度。
- 16-19 将来自第12章的**NestedSmartPointer.cpp**中的**ObjContainer**翻译成一个模板。用两个不同的类测试它。
- 16-20 通过模板化**class Stack**来修改**C15:OStack.h**和**C15:OStackTest.cpp**，使得它自动地从被包含类和从**Object**多重继承。被产生的**Stack**应当只接受和生成被包含类型的指针。

- 16-21 使用**vector**而不使用**Stack**重复练习20。
- 16-22 从**vector<void\*>**继承一个类**StringVector**，并且重新定义**push\_back()**和**operator[]**成员函数，使得它只接受和生成**string\***（并执行适当的类型转换）。现在，创建一个模板，它将自动地产生一个容器类以便对任何类型的指针做同样的事情。这个技术常常用于减少代码膨胀，防止过多的模板实例化。
- 16-23 在**TPStash2.h**中，对**PStash::iterator**添加和测试**operator-**，仿照**operator+**的逻辑。
- 16-24 在**Drawing.cpp**中，添加和测试一个函数模板，用来调用**erase()**成员函数。
- 16-25 （高级）修改**TStack2.h**中的**Stack**类以允许所有权的所有粒度：为每一个链表增加一个标志以表明它是否拥有其指向的对象，并在**push()**函数和析构函数中支持这一信息。增加用于读取和改变每一个链表所有权的成员函数。
- 16-26 （高级）修改来自第12章的**PointerToMemberOperator.cpp**，使得**FunctionObject**和**operator->\***被模板化，以便与任何返回类型工作（对于**operator->\***，必须用成员模板，这将在第2卷中介绍）。在**Dog**成员函数中，添加和测试对于零个、一个和两个参数的支持。

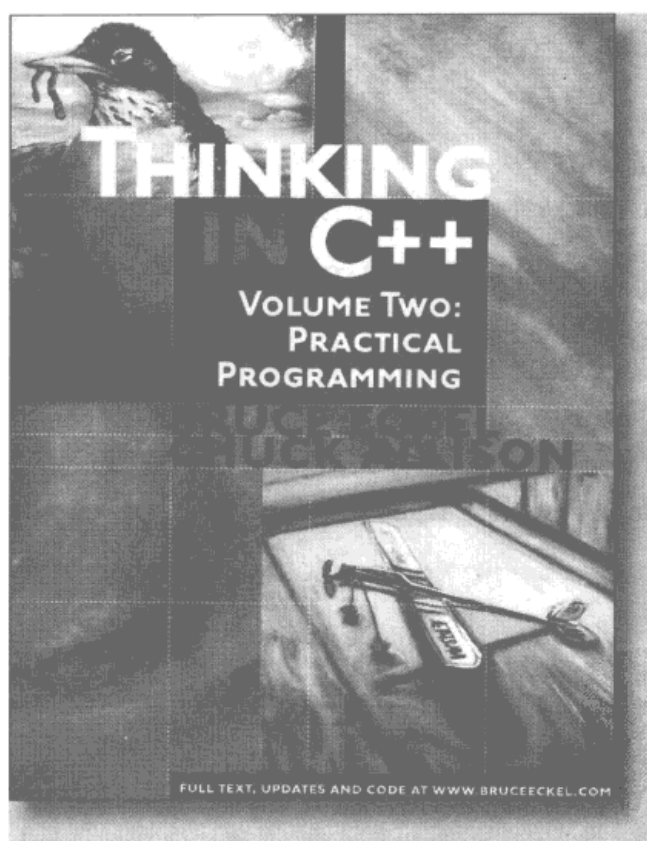




# 第2卷 实用编程技术

## Volume Two: Practical Programming

(美) Bruce Eckel Chuck Allison 著 刁成嘉 等译



电子书  
PDF  
PDG

“恭喜两位完成了这部经典之作！这部精品既妙趣横生，又不乏深度……所用专业知识的精确和语言应用的缜密真是让我大为震撼……我相信你们已经达到了大师级水平，简直太出色了！”

——《C/C++ Users Journal》杂志专栏主编 Bjorn Karlsson

“此书是一项巨大的成就，你的书架上早就该有这本书了。”

——《Doctor Dobbs Journal》杂志特约编辑 Al Stevens

“Eckel的作品是惟一一本如此清晰地阐述如何重新思考以面向对象方法构造程序的书籍。这本书也是一本讲授C++来龙去脉的优秀指南。”

——《Unix Review》杂志的编辑 Andrew Binstock

“Bruce 在C++方面的洞察力一次次令我惊叹，而这本《C++编程思想》则是他思想的精粹。如果你想获得C++中难题的清晰解答，就请购买这部杰作吧。”

——《The Tao of Objects》一书的作者 Gary Entsminger

“《C++编程思想》不仅系统而详细地探讨了何时和如何使用内联、引用、运算符重载、继承和动态对象等方面的重要问题，而且还讨论了一些深入的技术，如怎样正确使用模板、异常及多重继承等。Eckel本人的面向对象和程序设计的思想也完全融入这部著作中。《C++编程思想》是每个C++开发人员案头必备之书，即每一位用C++开发重要软件的开发人员必须拥有的一本书。”

——《PC Magazine》杂志特约编辑 Richard Hale Shaw



C++语言是一种使用广泛的程序设计语言，掌握了C++基础知识和基本编程技巧的人们，如果还想对C++有深入的了解，并且掌握更高级的C++编程技术的话，我们愿意向广大读者推荐《C++编程思想 第2卷：实用编程技术》的中译本。作者Bruce Eckel是C++标准委员会拥有表决权的成员之一，本书第1版荣获《软件开发》杂志评选的1996年度图书震撼大奖（Jolt Award），成为该年度最佳图书，在美国非常畅销。本书内容十分丰富，结构设计循序渐进，案例翔实而深入浅出，有一定的深度和广度。

二位作者致力于计算机教学数十年，经验十分丰富。在本书的讲授方法、例子和每章后面的练习的选用上都别具特色。通过一些非常简单的例子和简练的叙述，准确地阐明了C++编程实践中最困难的一些问题和概念，给人以拨云见日、耳目一新的感觉。读者在学习那些原本难于理解的内容时，常常会有豁然开朗的奇特效果，从而在不知不觉中接受并掌握了实用的编程技术。

本书介绍了实用的编程技术和最佳的实践方法，解决了C++开发中最困难的课题。内容上分为3部分：第一部分深入探究异常处理方法，清晰解释了异常安全设计；第二部分研究了C++的字符串、输入输出流、STL算法和容器，详细阐述了模板的现代用法，包括模板元编程；第三部分解释多重继承的难点，展示RTTI的实际使用，描述典型的设计模式及其实现，介绍被认为是标准C++下一版特征之一的多线程处理编程技术，并提供了最新的研究成果。书中所举的程序例子都经过多个软件平台和编译器的测试，稳定可靠。本书不仅适合C++的初学者，对有经验的C++程序员来说，每次阅读也总会有新的体会，这正是本书的魅力所在。也正因为如此，本书不仅适合作为高等院校计算机、信息技术及相关专业本科生、研究生的教材，也可供广大从事软件开发的研究人员和科技工作者参考。

作为译者，我早已耳闻《C++编程思想》是一本别具特色的畅销书，并拜读了本书第1卷的中译本，其内容、讲授方法和特色让我受益匪浅。受机械工业出版社华章公司的委托，我有幸承担《C++编程思想》第2卷的翻译工作。翻译这样的成功之作，既是机遇，又是挑战。在翻译的过程中惟恐因水平有限而不能将原著中精彩内容如实转达，所以在翻译本书的过程中力求忠于原著，对书中出现的大量专业术语力求遵循标准译法，并在有可能引起歧义的地方注上英文原文。

本书在翻译过程中受到南开大学信息学院计算机系刘璟教授的关心和支持，特此表示感谢。邢恩军、刘胜斐、罗仕波、郑莹莹、肖鹏、程玉鹏、黄硕、金士英、杨鹏飞、赵建树、田新、漆芳敏、费志泉、郜业军、申芳、杨志真、刁奕、高建国、旷昊、蓝炳伟、王叙、李平参加了本书部分章节的初译。由于水平有限，翻译不妥或错误之处在所难免，敬请广大读者批评指正。

通过对本套教材第1卷的学习，读者已经掌握了C与C++的基础知识。这一卷将涉及其更为高级的特性，使读者领悟C++编程的方法与思想，从而编写出健壮的C++程序。

现在假定读者已经熟悉了第1卷的内容。

## 目标

编写这套教材的目标是：

1. 每节只介绍适当的学习内容，使学习向前推进一小步。因此读者能很容易地在继续下一步学习前消化每个已学过的概念。

2. 讲授实用编程技巧，以便读者在日常的学习和工作中使用这些技巧。

3. 只把对于理解这门语言比较重要的内容介绍给读者，而不是将我们所知的一切都罗列出来。我们相信，不同信息的重要性是不同的。有些内容对于95%的程序员来说肯定没有必要知道，这些信息只会迷惑人们，加深人们对这门语言复杂性的恐惧。举一个关于C语言的例子，如果记住运算符优先级表（我们从未做到这一点），就能够写出漂亮的代码。但如果对其进行深究，它会让代码的读者或维护者感到迷茫。所以可以摒弃优先级，而在优先级不很清楚的情况下使用括号。同样，C++语言中的某些信息对于写编译程序的人员来说更为重要，而对程序员来说却没那么重要。

4. 尽可能将每一节内容充分集中，使得授课时间及两个练习之间的间隔时间不长。这样不仅能使读者的思维在每次课堂研讨会期间更加活跃与投入，还可使他们有更大的成就感。

5. 尽力不用任何特定厂商的C++版本。我们已在所有能见到的C++实现版本中测试了本教材中的代码（前言中稍后将有介绍），有的实现版本无法工作，那是因为它没有遵循C++标准，我们已经在示例中标注这些事实（读者会在源代码中看到这些标注），以便将其从构建过程中摒弃。

6. 教材中代码的自动编译和测试。由于已经发现未经编译和测试的代码很可能有问题，所以在这一卷中，本教材所提供的例子全是测试过的代码。此外，读者可从<http://www.MindView.net>下载这些代码，它们是直接从本教材的文本中摘录的，这些程序能够用自动生成的测试文件进行编译和运行测试。读者可以通过这种方式知道教材中的代码都是正确的。

## 各章简介

下面是本教材各章内容的简要介绍。

### 第一部分 建立稳定的系统

**第1章 异常处理。**出错处理在程序设计中一直是一个问题。即便你返回了错误信息或设置了一个标志，函数调用者还会对此视而不见。异常处理是C++的主要特征之一，该机制解

决这类问题的方法如下：在致命错误发生时，允许该函数“抛出”一个对象。对应于不同的错误抛掷不同类型的对象，那么该函数的调用者就可以在独立的出错处理子程序中“捕获”这些对象。如果程序中抛出了一个异常，该异常就不能被忽略，这样就可以保证会触发一些事件来响应这一错误。决定采用异常处理机制是影响代码设计向良性方向发展的重要方法。

**第2章 防御性编程。**许多软件故障都是可以预防的。防御性编程是一种编写代码的方式，采用此种方式能够较早地发现并更正错误，从而避免了这些错误对相关工作区域造成的危害。在开发过程中使用断言（assertion）是一种很重要的方法，该方法能够在程序员编写代码的过程中进行合法性检验，与此同时在代码中留下了一个可执行文档，该文档可用来揭示程序员开始编写代码时的思路。在向用户交出程序前应严格地测试编写的代码。对于成功地进行常规软件开发的人员来说，自动单元测试框架是一个不可缺少的工具。

## 第二部分 标准C++库

**第3章 深入理解字符串。**最为常见的编程工作是对文本进行处理。C++字符串类将程序员从内存管理事务中解脱出来，使其有足够的时间和精力增强文本处理能力。此外，为适应国际化应用的需求，C++也支持对宽字符和区域字符的操作。

**第4章 输入输出流。**输入输出流类是最早的C++库之一，它提供必不可少的输入输出功能。使用输入输出流类就是用I/O库来代替C语言中的`stdio.h`。这种I/O库用起来更容易、更灵活并且更易于扩展——可对其做适当的调整使之能够与新定义的类一起工作。该章告诉读者怎样充分利用现有的输入输出流类库来实现标准I/O、文件I/O以及内存中的格式化操作。

**第5章 深入理解模板。**现代C++的显著特征是模板的强大功能。模板的作用不仅仅在于生成容器。借助于模板，还可开发出具有健壮性、通用性和高性能的类库。关于模板的内容，需要了解的还有很多，它们构成了C++语言内的一个子语言，使得程序员能在更大程度上控制编译过程。模板的引入对C++程序设计来说是一场革命，可以毫不夸张地说，自从有了模板，C++程序设计焕然一新了。

**第6章 通用算法。**算法处于计算的核心。C++借助其模板功能提供了一大批功能强大、高效且易用的通用算法。标准算法也可以通过函数对象进行自定义。该章研究了模板库中的所有算法。（第6章和第7章讲的是标准C++模板库，也就是通常所说的标准模板库（Standard Template Library, STL）。）

**第7章 通用容器。**C++以一种类型安全的方式提供对所有常见数据结构的支持。用户不必为容器中的内容而感到忧虑，其对象的同一性得到了保证。可通过迭代器将容器的遍历与容器自身相分离，这是模板的又一杰作。这种巧妙的安排能够将算法灵活应用于容器，而容器则采用了最简单的设计。

## 第三部分 专题

**第8章 运行时类型识别。**当你只用一个对象指针或引用指向基类型时，运行时类型识别（RunTime Type Identification, RTTI）就会找到该对象的确切类型。一般情况下，有时会有意忽略掉一个对象的确切类型，而利用虚函数机制实现对应于那个类型的正确操作。但有时（比如当编写像调试器这样的软件工具时）借助于此信息知道一个对象的确切类型是非常有用的，常常可以非常有效地进行某些特殊操作。这一章解释RTTI的用途及其使用方法。

**第9章 多重继承。**一个新类可以从多个现存类中继承，这话乍听起来很简单。但是，由



此而产生的二义性和对基类对象的多次复制将很难避免。这些问题可通过建立虚基类来解决，但更大的问题仍然存在：什么时候用多重继承？只有当你需要通过多于一个的公共基类来操作一个对象时，多重继承才是必需的。这一章对多重继承的语法做了解释，也提出了可选方案——特别针对使用模板怎样解决一个典型问题进行了深入讨论。运用多重继承来修复一个“被损坏了的”类接口是关于C++这一特性的经典案例。

**第10章 设计模式。**自从对象产生以来，在程序设计领域最具革命性的飞跃是设计模式的引进。设计模式是对应于公认的编程问题的经典解决方案，它独立于语言之外，其表述方式的特殊性使它能应用于许多情况之下。因此，像单件（Singleton）、工厂方法（Factory Method）和访问者（Visitor）这样的模式现都已被一般的程序员接受和使用了。这一章介绍如何通过C++来实现和使用一些较为有用的设计模式。

**第11章 并发。**人们越来越期待有响应功能的用户接口，而这种接口能（看起来像）同时处理多任务。现代操作系统允许进程拥有共享进程地址空间的多线程。多线程程序设计要求编程人员有与众不同的思维方式，然而，在进行多线程程序设计时也会遇到一些困难。这一章通过一个可免费获得的类库（由IBM的 Eric Crahen 提供的ZThread库）介绍怎样使用C++来有效地管理多线程应用。

## 练习

我们发现，在课堂讨论期间使用简单的练习特别有助于学生对相关概念的理解。所以，在每一章后面都附有一定量的练习题。

这些练习题十分简单，可当堂完成；但有一点，需要有老师在场观察证实，以确保所有的学生都掌握了相关内容。有些练习题有一定的挑战性，是为激发优秀学生的学习兴趣准备的。所有练习被设计为可以在短时间内完成，只是用来测试和完善学生所掌握的知识，而不是为了提出挑战（很可能读者自己会找到这些难题——或者更可能的是难题会自己找上门来）。

## 源代码

本教材的源代码是免费版权软件，通过网站<http://www.MindView.net>发布。该版权是为了防止在未经许可的情况下在印刷媒体上再度出版这些代码。

只要遵守代码中的版权声明，读者就可以在自己的项目里和课堂上使用这些代码。

## 编译器

读者使用的编译器可能不支持本教材所论及的C++的所有特征，尤其是当该编译器并非是其最新版本的时候，这种情况就显得尤为突出。所以实现像C++这样的语言绝非易事；同时读者会希望C++的特征一点点展现，而非一下子全部出现。但是，如果读者试做了教材中的一个例子，结果编译器报告了一大堆错误，这就不一定仅仅是代码或编译器中的一个故障那么简单了——很可能在读者选用的编译器上根本就运行不了那个代码。

教材中的代码已经用很多编译器进行过测试，目的是确保这些代码符合C++标准，并且在尽可能多的编译器上运行。遗憾的是，并非所有的编译器都符合C++标准，因此在使用这些编译器构造可执行文件时，去掉了某些文件。这些被去除的文件在makefiles里都有体现，而makefiles是为这本教材的代码包自动生成的，并且可从<http://www.MindView.net>下载。在makefiles中，从每个程序代码清单开头的注释中都可以看到这些嵌入的排除标记符，这样读

者将会知道是否应让某个特定的编译器来运行那个代码（少数情况下，编译器确实会编译代码，但执行动作却是错的，本教材将这些代码也排除在外）。

下面就是有关的排除标记符和相应的编译器：

- **{-dmc}** Walter Bright的Digital Mars编译器，专为Windows设计，可从[www.DigitalMars.com](http://www.DigitalMars.com)免费下载。这种编译器兼容性超强，整本教材中几乎都看不到该排除标记符。
- **{-g++}** 免费的Gnu C++ 3.3.1，在大多数Linux软件包和Macintosh OSX中都预装了该编译器。该编译器也是专为Windows设计的Cygwin的一部分（见下文）。从[gcc.gnu.org](http://gcc.gnu.org)可以得到其为其他大多数操作平台而设计的版本。
- **{-msc}** Visual C++.NET是微软(Microsoft)推出的第7版编译器（使用前必须先安装Visual Studio.NET，不能免费下载）。
- **{-bor}** Borland C++第6版（不可免费下载；这是最新的版本）。
- **{-edg}** Edison Design Group (EDG) C++。该编译器可用来检测代码是否符合标准C++。只是由于类库的原因才出现了这个标记符，因为本教材采用了Dinkumware有限公司赠送的带有兼容类库的EDG前端免费副本。单独使用编译器不会出现任何编译错误。
- **{-mwcc}** 为Macintosh OSX设计的Metrowerks Code Warrior。注意，使用OSX也必须先安装Gnu C++编译器。

如果从<http://www.MindView.net>下载并解压了本教材的代码包，读者会发现用来为上述编译器建立代码的构造文件（makefiles）。本教材使用免费的GNU-**make**，它在Linux、Cygwin（一个可在Windows上运行的免费Unix shell，详见[www.Cygwin.com](http://www.Cygwin.com)）环境下运行，也可安装在读者自己的计算机平台上——详见[www.gnu.org/software/make](http://www.gnu.org/software/make)。（这些文件在其他**make**上也可能运行，但得不到支持。）一旦安装了**make**，如果在命令行运行方式下键入**make**，就会得到有关如何为上述编译器建立教材中代码的操作步骤。

注意，本教材程序示例文件中的这些标记符指出了当时调试用的编译器的版本。很可能在这本教材出版后相应的编译器版本已经升级。也有可能我们在使用很多编译器对本教材中的代码进行编译时，错误地配置了某个编译器；如果没有配错编译器，则相应的代码应该早已经被正确地编译。因此，在自己的编译器上重新调试这些代码，并检查从<http://www.MindView.net>网站下载的代码是否为最新版本就显得尤为重要。

## 语言标准

在本教材中，当提到ANSI/ISO C标准时，指的是1989标准，而且一般情况下只是说“C”。只有当有必要区分标准1989 C和较早的版本，如制定标准前的C语言版本时，才会做出区分。在本教材中并不涉及C99。

ANSI/ISO C++委员会很早以前就制定出了第一个C++标准，通常称为C++98。本教材用“标准C++”来指这个标准化语言。如果只说C++，那就意味着是“标准 C++”。C++标准委员会还在继续发布对使用C++的公众群体很重要的信息，这些会促使另一C++标准C++0x的形成，但它的产生在近几年内不太可能实现。

## 研讨班和咨询

Bruce Eckel的公司——MindView公司，提供基于本教材中的材料和高级主题的公共实习

培训研讨班。每课所讲的都是从各章中精选的内容，每次讲授完毕，后面有一个检测练习阶段，每个学生都能够受到个别指导。我们还提供现场培训、咨询、辅导、设计和代码演练的服务。从<http://www.MindView.net>网站上可获得有关即将开办的研讨班信息、相关报名表和其他联系信息。

## 错误

无论我们怎样挖空心思地去检查错误，总会漏掉一些错误，但这些错误却常常能被热心的读者发现。如果读者发现了任何认为是错误的地方，请使用本教材电子版中的反馈系统与我们联系。读者可在<http://www.MindView.net>网站上找到该系统。非常感谢您的帮助。



# 第一部分 建立稳定的系统

通常，软件工程师花费在检查代码方面的时间同编写代码所花费的时间相当。保证软件的质量是每个程序员追求的目标，在问题出现之前将其消灭对程序员实现这个目标大有裨益。另外，软件系统应具有在不可预测的环境中正常运行的能力。

C++中引入了异常处理机制来支持复杂的出错处理，从而避免大量的出错处理逻辑干扰程序的代码。第1章介绍恰当地使用异常处理对性能良好的软件是何等的重要，该章还介绍了建立在异常安全代码基础之上的设计原则。第2章涉及单元测试和调试技术，意在使程序在其发布之前质量尽可能地好。使用断言（assertion）来表示和强化程序中的不变量（invariant），是经验丰富的软件工程师的确切标志。此外，本章还将介绍一个支持单元测试的简单框架。



## 异常处理

增强错误恢复能力是提高代码健壮性的最有力的途径之一。

遗憾的是，在实践中人们通常会忽略出错情况，就好像程序处在一个无错误的状态下进行工作。毫无疑问，导致上述问题的一个原因就是，检测错误是一个乏味的工作并且会导致代码的膨胀。比如，函数**printf()**返回那些被成功地打印出来的字符的个数，但是却很少有人去检测这个返回值。单单代码激增一项就足以令人厌恶，更不用说代码膨胀将不可避免地增加程序阅读的困难了。

C语言中采用的出错处理方法被认为是“紧耦合的”——函数的使用者必须在非常靠近函数调用的地方编写错误处理代码，这样会使其变得笨拙和难以使用。

异常处理(exception handling)是C++的主要特征之一，是考虑问题和处理错误的一种更好的方式。使用异常处理：

1) 错误处理代码的编写不再冗长乏味，并且不再与“正常的”代码混合在一起。程序员只需编写希望产生的代码，然后在后面的某个单独的区段里编写处理错误的代码。如果要多次调用同一个函数，则只需在某个地方编写一次错误处理代码。

2) 错误不能被忽略。如果一个函数必须向调用者发送一条错误消息，它将“抛出”一个描述这个错误的对象。如果调用者没有“捕获”并处理它，错误对象将进入上一层封装的动态范围，并且一直继续下去，直到该错误被捕获或者因为程序中没有异常处理器捕获这种类型的异常而导致程序终止。

本章将分析C中的错误处理方法，并讨论为什么该方法在C中工作得不尽如人意，以及为什么在C++中根本无法使用该方法。本章还介绍了**try**、**throw**和**catch**等在C++中用于支持异常处理的关键字。

### 1.1 传统的错误处理

在本教材的大多数例子中，我们使用**assert()**的意图是：用于开发阶段的调试，通过宏定义语句**#define NDEBUG**使其在最终发行的软件产品中失效。运行时错误检测处理采用了在第1卷第9章中开发的**require.h**中定义的函数(**assure()**和**require()**)，该文件也附在本教材的附录B中。这些函数以一种方便的方式表达了这样的意思，“这儿发生了一个错误，读者可能希望用更复杂的代码来处理该错误，但是在本例中却不必对此过多地分心”。**require.h**中定义的函数对于一些小的程序来说已经足够了，但是对于更复杂的应用则应当采用更加完善的错误处理代码。

当确切地知道应该做什么的时候，错误处理将会非常地简单明了，因为在语境中已经知道了所有必要的信息。程序员可以在错误发生的时候立即处理它。

当在某个语境中不能得到足够的信息时，问题就出现了，这时需要将错误信息传递到一个包含上述信息的不同的语境中去。在C语言中，有三种方法处理这种情况：

1) 在函数中返回错误信息，如果无法将返回值用于这个方面，则设置一个全局的错误状态标志(标准C中提供了**errno**和**perror()**来支持这种方法)。就像前面提到的一样，程序员会因为冗长乏味的错误检查而倾向于忽略错误信息。另外，从一个出现异常情况的函数中返回可

能根本没有意义。

2) 使用鲜为人知的标准C库中的信号处理系统,由函数**signal()** (用以推断事件发生时出现了什么情况)和函数**raise()** (产生一个事件)实现。同样,该方式的耦合度非常高,因为如果要使用可能产生信号的库函数,库的使用者必须了解和设置适当的信号处理机制。在大型项目中,不同的库产生的信号值可能会发生冲突。

3) 使用标准C库中的非局部跳转函数:**setjmp()**和**longjmp()**。使用**setjmp()**可以在程序中保存一个已知的无错误的状态,一旦发生错误,就可以通过调用**longjmp()**返回到该状态。同样,这使得错误发生的位置与保存状态的位置之间产生高度耦合。

当考虑C++的错误处理方案时,会涉及另一个关键问题:C中的信号处理方法和函数**setjmp()/longjmp()**并不调用析构函数,所以对象不会被正确地清理。(实际上,如果**longjmp()**跳转到超出析构函数的作用范围以外的地方,则程序的行为是不可预料的)。在这种情况下不可能有效地从错误状态中恢复,因为程序中留下了未被清理但又不能被再次访问的对象。下面的代码演示了函数**setjmp()/longjmp()**的使用方法:

```
//: C01:Nonlocal.cpp
// setjmp() & longjmp().
#include <iostream>
#include <setjmp>
using namespace std;

class Rainbow {
public:
    Rainbow() { cout << "Rainbow()" << endl; }
    ~Rainbow() { cout << "~Rainbow()" << endl; }
};

jmp_buf kansas;

void oz() {
    Rainbow rb;
    for(int i = 0; i < 3; i++)
        cout << "there's no place like home" << endl;
    longjmp(kansas, 47);
}

int main() {
    if(setjmp(kansas) == 0) {
        cout << "tornado, witch, munchkins..." << endl;
        oz();
    } else {
        cout << "Auntie Em! "
              << "I had the strangest dream..."
              << endl;
    }
}
//:~
```

函数**setjmp()**的行为很特别,因为如果直接调用它,它便会将所有与当前处理器相关的状态信息(比如指令指针的内容、运行时栈指针等)保存到**jmp\_buf**中去并返回0。在这种情况下,它的表现与一个普通的函数一样。然而,如果使用同一个**jmp\_buf**调用**longjmp()**,则函数返回时就好像刚从**setjmp()**中返回时一样——又回到了刚刚从**setjmp()**返回的地方。这一次,返回值是调用**longjmp()**时所使用的第二个参数(argument),因此可以通过这个值断定程序实际是从**longjmp()**返回的。读者可以想象有很多不同的**jmp\_buf**的情况,程序可以跳到多个不同的位置。局部**goto** (使用标号)和这种非局部跳转的区别在于,通过**setjmp()/**

**longjmp()** 程序可以返回到运行栈中任何预先确定的位置（即任何调用 **setjmp()** 的位置）。

在C++中使用 **longjmp()** 的问题是它不能识别对象。特别是，当跳出某个作用域的时候，它不会调用对象的析构函数<sup>①</sup>。而析构函数的调用在C++中是必需的操作，所以这种方法不能用于C++。实际上，C++标准已经说明，使用 **goto** 跳入某个作用域（有效地跳过构造函数的调用），或使用 **longjmp()** 跳出某个作用域而且这个作用域的栈中有某个对象需要析构时，程序的行为是不确定的。

## 1.2 抛出异常

如果在程序的代码中出现了异常的情况——也就是说，通过当前语境无法获得足够的信息以决定应该采取什么样的措施——程序员可以创建一个包含错误信息的对象并把它抛出当前语境，通过这种方式将错误信息发送到更大范围的语境中。这种方式被称为“抛出一个异常”。如下所示：

```
//: C01:MyError.cpp {RunByHand}

class MyError {
    const char* const data;
public:
    MyError(const char* const msg = 0) : data(msg) {}
};

void f() {
    // Here we "throw" an exception object:
    throw MyError("something bad happened");
}

int main() {
    // As you'll see shortly, we'll want a "try block" here:
    f();
} ///:~
```

在上面的代码中，**MyError** 是一个普通的类，它的构造函数接受一个 **char\*** 类型的变量作为参数。在抛出一个异常时，可以使用任意的类型（包括内置类型），但通常应当为抛出的异常创建特定的类。

关键字 **throw** 将导致一系列事情的发生。首先，它将创建程序所抛出的对象的一个拷贝，然后，实际上，包含 **throw** 表达式的函数返回了这个对象，即使该函数原先并未设计为返回这种对象类型。一种简单的考虑异常处理的方式是将其看做是一种交错返回机制（alternate return mechanism）（如果仔细分析这个问题，就会发现自己陷入了困境）。当然也可以通过抛出一个异常而离开正常的作用域。在任何一种情况下都会返回一个值，并且退出函数或作用域。

由于异常会造成程序返回到某个地方，而这个地方与正常函数调用时遇到 **return** 语句后程序返回到的地方完全不同，所以异常与 **return** 语句的相似性仅止于此。（可以在程序中恰当的部分编写异常处理器代码，这段代码可能与异常抛出的位置相差很远。）另外，异常发生之前创建的局部对象被销毁。这种对局部对象的自动清理通常被称为“栈反解（stack unwinding）”。

而且，在程序中可以抛出许多程序员希望的不同类型的对象。典型的做法是，程序员要为

① 当读者运行这个例子的时候，可能会感到奇怪——某些C++编译器包含了扩展的 **longjmp()**，能够清除栈中的对象。但这种行为是不可移植的。

每一种不同的异常情况抛出不同类型的对象。这么做的目的是将错误信息保存在相应的对象和对象类名中，这样，在调用者的语境中根据这些信息就可以决定应该如何处理这些异常了。

### 1.3 捕获异常

就像前面提到的一样，C++异常处理机制的一个好处是，可以使程序员在一个地方专注于所要解决的问题，而在另一个地方对这段代码所产生的错误进行处理。

#### 1.3.1 try块

如果在一个函数内部抛出了异常（或者被这个函数所调用的其他函数抛出了异常），这个函数将会因为抛出异常而退出。如果不想因为一个**throw**而退出函数，可以在函数中试图解决实际产生程序设计问题的地方（和可能产生异常的地方）设置一个**try**块。这个块被称做**try**块的原因是程序需要在这里尝试着调用各种函数。**try**块只是一个普通的程序块，由关键字**try**引导：

```
try {
    // Code that may generate exceptions
}
```

如果希望通过仔细地检查每一个被调用函数的返回值来发现错误，程序员必须围绕每一次函数调用编写初始化和检测代码，即使多次调用同一个函数也是如此。使用异常处理时，可以将所有工作放入**try**块中，然后在**try**块的后面处理可能产生的异常。这样一来，代码将更容易编写和阅读，因为代码的设计目标不会被错误处理所干扰。

#### 1.3.2 异常处理器

当然，被抛出的异常肯定会在某个地方终止。这个地方就是异常处理器（exception handler），程序员需要为每一种想捕获的异常类型设置一个异常处理器。然而，多态对于异常同样有效，因此一个异常处理器可以处理某种异常类型和这种类型的派生类。

异常处理器紧接着**try**块，并且由关键字**catch**标识：

```
try {
    // Code that may generate exceptions
} catch(type1 id1) {
    // Handle exceptions of type1
} catch(type2 id2) {
    // Handle exceptions of type2
} catch(type3 id3)
    // Etc...
} catch(typeN idN)
    // Handle exceptions of typeN
}
// Normal execution resumes here...
```

**catch**子句的语法类似于带有单一参数的函数。可以在异常处理器内部使用标识符（**id1**、**id2**等），就像使用函数的参数一样。如果不需要在异常处理器中使用标识符，这些标识符可以省略。异常类型通常提供了对其进行处理的足够的信息。

异常处理器都必须紧跟在**try**块之后。一旦某个异常被抛出，异常处理机制将会依次寻找参数类型与异常类型相匹配的异常处理器。找到第一个这样的异常处理器后，程序的执行流程进入这个**catch**子句，于是系统就可以认为该异常已经处理了。（查找异常处理器的过程在找到第一个匹配的**catch**子句之后就终止了。）只有匹配的**catch**子句才会执行；在执行完最后一个与该**try**块相关的异常处理器后，程序又恢复到正常的控制流程。



注意，在**try**块中，不同的函数调用可能产生相同类型的异常，这时，只需要一个异常处理器就可以了。

为了举例说明**try**和**catch**，我们在这里修改了**Nonlocal.cpp**，将其中的**setjmp()**用一个**try**块代替，将**longjmp()**用一个**throw**语句代替：

```
//: C01:Nonlocal2.cpp
// Illustrates exceptions.
#include <iostream>
using namespace std;

class Rainbow {
public:
    Rainbow() { cout << "Rainbow()" << endl; }
    ~Rainbow() { cout << "~Rainbow()" << endl; }
};

void oz() {
    Rainbow rb;
    for(int i = 0; i < 3; i++)
        cout << "there's no place like home" << endl;
    throw 47;
}

int main() {
    try {
        cout << "tornado, witch, munchkins..." << endl;
        oz();
    } catch(int) {
        cout << "Auntie Em! I had the strangest dream..."
            << endl;
    }
} //:~
```

当执行函数**oz()**中的**throw**语句时，程序的控制流程开始回溯，直到找到某个带有**int**型参数的**catch**子句为止。程序在这个**catch**子句的主体中恢复运行。这个程序与**Nonlocal.cpp**的最重要的区别在于，当**throw**语句造成程序的执行过程从**oz()**函数返回时，对象**rb**的析构函数被调用。

### 1.3.3 终止和恢复

在异常处理理论中有两个基本的模型：终止和恢复。在终止（termination）（C++支持这种模型）模型中，假定错误非常严重，以至于不可能在异常发生的地点自动恢复程序的执行。也就是说，无论谁抛出一个异常，都表明程序已经陷入了无法挽救的困境，并且不需要再返回发生异常的地方。

另一个异常处理模型被称为恢复（resumption）模型，在20世纪60年代，PL/I语言首先引入该模型<sup>①</sup>。使用恢复模型意味着异常处理器希望能够校正这种情况，然后自动地重新执行发生错误的代码，并希望第二次执行能够成功。如果希望在C++中重新恢复程序的执行，则必须显式地将程序的执行流程转移到错误发生的地方，通常的做法是重新调用发生错误的函数。将**try**块放到**while**循环中，不断地重新执行**try**块中的程序，直到产生满意的结果，这种做法并不罕见。

历史上，使用支持恢复性异常处理模型的操作系统的程序员们，最终使用的是类似终止模

<sup>①</sup> 通过ON ERROR功能，BASIC语言长期以来支持一种有限形式的恢复性异常处理模型。

型的代码并跳过了异常恢复。虽然恢复模型非常吸引人，但是在实践中并没有多大用处。其中一个原因或许就是发生异常的位置和异常处理器之间的距离。对于远处的异常处理器来说，终止执行是一个问题；而另一个问题是，对于一个大型系统来说，在很多位置都可能发生异常，从异常发生的位置跳转到远处的异常处理器然后再返回，这在概念上也十分困难。

## 1.4 异常匹配

一个异常被抛出以后，异常处理系统将按照在源代码中出现的顺序查找最近的异常处理器。一旦找到匹配的异常处理器，就认为该异常已经被处理了而不再继续查找下去。

匹配一个异常并不要求异常与其处理器之间完全相关。一个对象或者是指向派生类对象的引用都会与其基类处理器匹配。（然而，如果异常处理器是针对对象而不是针对引用的，这个异常对象将会被“切割”——被截取成基类对象——就好像一个基类对象被传递给了异常处理器。除了丢失派生类包含的所有附加信息之外，这并没有什么危害。）由于这种原因，并且为了避免再次拷贝异常对象，最好是通过引用而不是通过值来捕获异常<sup>①</sup>。如果一个指针被抛出，将使用通常的标准指针转换来匹配异常。但是，在匹配过程中，不会将一种异常类型自动转换成另一种异常类型。比如：

```
//: C01:Autoexcp.cpp
// No matching conversions.
#include <iostream>
using namespace std;
class Except1 {};

class Except2 {
public:
    Except2(const Except1&) {}
};

void f() { throw Except1(); }

int main() {
    try { f();
    } catch(Except2&) {
        cout << "inside catch(Except2)" << endl;
    } catch(Except1&) {
        cout << "inside catch(Except1)" << endl;
    }
} ///:~
```

尽管读者可能会认为，通过使用转换构造函数（converting constructor）将一个**Except1**对象转换成一个**Except2**对象，可以使得第一个异常处理器被匹配。但是，异常处理系统在处理异常的过程中并不做这种转换，结果是，程序在**Except1**异常处理器那里结束。

下面的例子显示了基类的异常处理器怎样就能够捕获派生类异常：

```
//: C01:Basexcpt.cpp
// Exception hierarchies.
#include <iostream>
using namespace std;

class X {
```

① 读者可能总是希望在异常处理器中通过**const**引用来指定异常对象。（极少有程序在异常处理器中修改异常和重新抛出异常。）我们不对这种做法武断地评价。

```

public:
    class Trouble {};
    class Small : public Trouble {};
    class Big : public Trouble {};
    void f() { throw Big(); }
};

int main() {
    X x;
    try {
        x.f();
    } catch(X::Trouble&) {
        cout << "caught Trouble" << endl;
        // Hidden by previous handler:
    } catch(X::Small&) {
        cout << "caught Small Trouble" << endl;
    } catch(X::Big&) {
        cout << "caught Big Trouble" << endl;
    }
} ///:~

```

在这里，异常处理机制总是将**Trouble**对象，或派生自**Trouble**的任何对象（通过公有继承）<sup>①</sup>，匹配到第一个异常处理器。由于第一个异常处理器截获了所有的异常，所以第二和第三个异常处理器永远不会被调用。比较有意义的做法是，首先捕获派生类异常，并且将基类放到最后用于捕获其他不太具体的异常。

需要注意的是，这些例子都是通过引用来捕获异常的，尽管对这些类而言这一点并不重要，因为派生类中没有附加的成员，而且异常处理器中也没有参数标识符。通常情况下，应该在异常处理器中使用引用参数而不是值参数，以防异常对象所包含的信息被切割掉。

#### 1.4.1 捕获所有异常

有时候，程序员可能希望创建一个异常处理器，使其能够捕获所有类型的异常。用省略号代替异常处理器的参数列表就可以实现这一点：

```

catch(...) {
    cout << "an exception was thrown" << endl;
}

```

由于省略号异常处理器能够捕获任何类型的异常，所以最好将它放在异常处理器列表的最后，从而避免架空它后面的异常处理器。

省略号异常处理器不允许接受任何参数，所以无法得到任何有关异常的信息，也无法知道异常的类型。它是一个“全能捕获者”。这种**catch**子句经常用于清理资源并重新抛出所捕获的异常。

#### 1.4.2 重新抛出异常

当需要释放某些资源时，例如网络连接或位于堆上的内存需要释放时，通常希望重新抛出一个异常。（详见本章后面的“资源管理”一节。）如果发生了异常，读者不必关心到底是什么错误导致了异常的发生——只需要关闭以前打开的一个连接。此后，读者希望在某些更接近用户的语境（也就是说，在调用链中的更高层次）中对异常进行处理。在这种情况下，省略号异常处理器正符合这样的要求。这种处理方法，可以捕获所有异常，清理相关资源，然后重新抛

① 只有明确的、可访问的基类才能够捕获派生类异常。这种规则将验证异常所需的运行代价减到最小。请记住，异常是在运行时而不是在编译时被检测的，因此，编译时存在的大量信息在处理异常的时候是不存在的。

出该异常，以使得其他地方的异常处理器能够处理该异常。在一个异常处理器内部，使用不带参数的**throw**语句可以重新抛出异常：

```
catch(...) {
    cout << "an exception was thrown" << endl;
    // Deallocate your resource here, and then rethrow
    throw;
}
```

与同一个**try**块相关的随后的**catch**子句仍然会被忽略——**throw**子句把这个异常传递给位于更高一层语境中的异常处理器。另外，这个异常对象的所有信息都会保留，所以位于更高一层语境中的捕获特定类型异常的异常处理器能够获取这个对象包含的所有信息。

### 1.4.3 不捕获异常

就像在这一章的开始所说的，因为异常不可以忽略，而且将错误处理逻辑从问题发生地附近分开，所以异常处理被认为比传统的返回错误代码的技术要好。如果**try**块之后的异常处理器不能匹配所抛出的异常，那么这个异常就会被传递给位于更高一层语境中的异常处理器，也就是说包含了不捕获这个异常的**try**块的函数或**try**块。（由于**try**块的位置处在函数调用链的较高层次，所以乍看起来并不明显。）这个过程持续进行，直到某一层存在一个异常处理器能够匹配这个异常。这时，异常处理系统认为已经“捕获”了这个异常，不再继续查找其他的异常处理器。

#### 1. **terminate()** 函数

如果没有任何一个层次的异常处理器能够捕获某种异常，一个特殊的库函数**terminate()**（在头文件<exception>中定义）会被自动调用。默认情况下，**terminate()**调用标准C库函数**abort()**使程序执行异常终止而退出。在Unix系统中，**abort()**还会导致主存储器信息转储（core dump）。当**abort()**被调用时，程序不会调用正常的终止函数，也就是说，全局对象和静态对象的析构函数不会执行。在下列两种情况下，**terminate()**函数也会执行：局部对象的析构函数抛出异常时，栈正在进行清理工作（也称栈反解，即异常的抛出过程被打断）；或者是全局对象或静态对象的构造函数或析构函数抛出一个异常。（一般来说，不允许析构函数抛出异常。）

#### 2. **set\_terminate()** 函数

通过使用标准的**set\_terminate()**函数，可以设置读者自己的**terminate()**函数，**set\_terminate()**返回被替换的指向**terminate()**函数的指针（第一次调用**set\_terminate()**函数时，返回函数库中默认的**terminate()**函数的指针），这样就可以在需要的时候恢复原来的**terminate()**。自定义的**terminate()**函数不能有参数，而且其返回值的类型必须是**void**。另外，这里所设置的**terminate()**函数不能返回（**return**）也不能抛出异常，而且它必须执行某种方式的程序终止逻辑。如果**terminate()**函数被调用，就意味着问题已经无法解决了。

下面的例子显示了如何使用**set\_terminate()**函数。在这个例子中，**set\_terminate()**函数的返回值被保存下来并且被还原，使得**terminate()**函数可以用来帮助隔离产生不可捕获的异常的代码块。

```
//: C01:Terminator.cpp
// Use of set_terminate(). Also shows uncaught exceptions.
#include <exception>
#include <iostream>
using namespace std;
```

```

void terminator() {
    cout << "I'll be back!" << endl;
    exit(0);
}
void (*old_terminate)() = set_terminate(terminator);

class Botch {
public:
    class Fruit {};
    void f() {
        cout << "Botch::f()" << endl;
        throw Fruit();
    }
    ~Botch() { throw 'c'; }
};

int main() {
    try {
        Botch b;
        b.f();
    } catch(...) {
        cout << "inside catch(...)" << endl;
    }
} ///:~

```

**old\_terminate**的定义乍看起来有点让人迷惑：它不但创建了一个指向函数的指针，而且用**set\_terminate()**函数的返回值初始化这个指针。尽管读者可能熟悉在指向函数的指针的声明之后紧跟一个分号的定义形式，但是，在这段代码中使用的恰好是另一种可以在定义时初始化的变量。

类**Botch**不仅在函数**f()**中抛出异常，而且在析构函数中也抛出异常。在**main()**函数中可以看出，正是析构函数中抛出的异常造成了程序调用**terminate()**。尽管异常处理器被声明成**catch(...)**，看起来应该能够捕获所有异常，不会导致对**terminate()**的调用，但是，实际上**terminate()**总会被调用。程序在处理一个异常的时候会释放在栈上分配的对象，这时，**Botch**的析构函数被调用，从而产生了第二个异常，这个异常迫使程序调用**terminate()**。因此，抛出异常或由于某种原因导致一个异常被抛出的析构函数通常被认为象征着拙劣的设计或糟糕的编码。

## 1.5 清理

异常处理的魅力之一在于程序能够从正常的处理流程中跳转到恰当的异常处理器中。如果异常抛出时，程序不做恰当的清理工作，那么异常处理本身并没有什么用处。C++的异常处理必须确保当程序的执行流程离开一个作用域的时候，对于属于这个作用域的所有由构造函数建立起来的对象，它们的析构函数一定会被调用。

这里有一个例子，演示了当构造函数没有正常结束时不会调用相关联的析构函数。这个例子还显示了当在创建对象数组的过程中抛出异常时会发生什么情况：

```

//: C01:Cleanup.cpp
// Exceptions clean up complete objects only.
#include <iostream>
using namespace std;

class Trace {
    static int counter;
    int objid;

```

```

public:
    Trace() {
        objid = counter++;
        cout << "constructing Trace #" << objid << endl;
        if(objid == 3) throw 3;
    }
    ~Trace() {
        cout << "destructing Trace #" << objid << endl;
    }
};

int Trace::counter = 0;

int main() {
    try {
        Trace n1;
        // Throws exception:
        Trace array[5];
        Trace n2; // Won't get here.
    } catch(int i) {
        cout << "caught " << i << endl;
    }
} ///:~

```

读者可以通过跟踪程序的执行过程来了解类**Trace**的对象踪迹。它用一个静态数据成员**counter**来统计已经创建的对象个数，而用普通数据成员**objid**来追踪特定对象的编号。

**main**函数首先创建一个单独的对象**n1** (**objid** 0)，然后试图创建一个具有五个**Trace**对象的数组，但是，在第四个对象 (#3) 被完整创建之前抛出了一个异常。对象**n2**根本就没有被创建。在这里可以看到程序的输出结果为：

```

constructing Trace #0
constructing Trace #1
constructing Trace #2
constructing Trace #3
destructing Trace #2
destructing Trace #1
destructing Trace #0
caught 3

```

对象数组的三个元素成功创建了，但是在创建第四个对象数组元素的过程中构造函数抛出了一个异常。在**main()**函数中，由于第四个构造函数（用于创建**array[2]**对象）没有完成，所以只有**array[1]** 对象和**array[0]**对象的析构函数被调用。最后，对象**n1**销毁。因为对象**n2**根本就没有创建，所以也没有销毁。

### 1.5.1 资源管理

当在编写的代码中用到异常时，非常重要的一点是，读者应该问一下，“如果异常发生，程序占用的资源都被正确地清理了吗？”大多数情况下不用担心，但是在构造函数里有一个特殊的问题：如果一个对象的构造函数在执行过程中抛出异常，那么这个对象的析构函数就不会被调用。因此，编写构造函数时，程序员必须特别的仔细。

困难的事情是在构造函数中分配资源。如果在构造函数中发生异常，析构函数将没有机会释放这些资源。这个问题经常伴随着“悬挂”指针 (“naked” pointer) 出现。例如：

```

//: C01:Rawp.cpp
// Naked pointers.
#include <iostream>
#include <cstdint>
using namespace std;

```

```

class Cat {
public:
    Cat() { cout << "Cat()" << endl; }
    ~Cat() { cout << "~Cat()" << endl; }
};

class Dog {
public:
    void* operator new(size_t sz) {
        cout << "allocating a Dog" << endl;
        throw 47;
    }
    void operator delete(void* p) {
        cout << "deallocating a Dog" << endl;
        ::operator delete(p);
    }
};

class UseResources {
    Cat* bp;
    Dog* op;
public:
    UseResources(int count = 1) {
        cout << "UseResources()" << endl;
        bp = new Cat[count];
        op = new Dog;
    }
    ~UseResources() {
        cout << "~UseResources()" << endl;
        delete [] bp; // Array delete
        delete op;
    }
};

int main() {
    try {
        UseResources ur(3);
    } catch(int) {
        cout << "inside handler" << endl;
    }
} ///:~

```

程序的输出为:

```

UseResources()
Cat()
Cat()
Cat()
allocating a Dog
inside handler

```

程序的执行流程进入了**UseResources**的构造函数，**Cat**的构造函数成功地完成了创建对象数组中的三个对象。然而，在**Dog::operator new()**函数中抛出了一个异常（用于模拟内存不足错误（out-of-memory error））。程序在执行异常处理器之时突然终止，**UseResources**的析构函数没有被调用。这是正确的，因为**UseResources**的构造函数没有完成，但是，这也意味着，在堆上成功创建的**Cat**对象不会被销毁。

### 1.5.2 使所有事物都成为对象

为了防止资源泄漏，读者必须使用下列两种方式之一来防止“不成熟的”的资源分配方式：

- 在构造函数中捕获异常，用于释放资源。
- 在对象的构造函数中分配资源，并且在对象的析构函数中释放资源。

使用下述方法可以使对象的每一次资源分配都具有原子性，由于资源分配成为局部对象生命周期的一部分，如果某次分配失败了，那么在栈反解的时候，其他已经获得所需资源的对象能够被恰当地清理。这种技术称为资源获得式初始化 (Resource Acquisition Is Initialization, RAII)，因为它使得对象对资源控制的时间与对象的生命周期相等。为了达到上述目标，利用模板修改前一个例子是一个好方法：

```
//: C01:Wrapped.cpp
// Safe, atomic pointers.
#include <iostream>
#include <cstdint>
using namespace std;

// Simplified. Yours may have other arguments.
template<class T, int sz = 1> class PWrap {
    T* ptr;
public:
    class RangeError {}; // Exception class
    PWrap() {
        ptr = new T[sz];
        cout << "PWrap constructor" << endl;
    }
    ~PWrap() {
        delete[] ptr;
        cout << "PWrap destructor" << endl;
    }
    T& operator[](int i) throw(RangeError) {
        if(i >= 0 && i < sz) return ptr[i];
        throw RangeError();
    }
};

class Cat {
public:
    Cat() { cout << "Cat()" << endl; }
    ~Cat() { cout << "~Cat()" << endl; }
    void g() {}
};

class Dog {
public:
    void* operator new[](size_t) {
        cout << "Allocating a Dog" << endl;
        throw 47;
    }
    void operator delete[](void* p) {
        cout << "Deallocating a Dog" << endl;
        ::operator delete[](p);
    }
};

class UseResources {
    PWrap<Cat, 3> cats;
    PWrap<Dog> dog;
public:
    UseResources() { cout << "UseResources()" << endl; }
    ~UseResources() { cout << "~UseResources()" << endl; }
    void f() { cats[1].g(); }
```





```
};

int main() {
    try {
        UseResources ur;
    } catch(int) {
        cout << "inside handler" << endl;
    } catch(...) {
        cout << "inside catch(...)" << endl;
    }
} //:-~
```

这种使用模板来封装指针的方法与第一种方法的区别在于，这种方法使得每个指针都被嵌入到对象中。在调用**UseResources**类的构造函数之前这些对象的构造函数首先被调用，并且如果它们之中的任何一个构造函数在抛出异常之前完成，那么这些对象的析构函数也会在栈反解的时候被调用。

**PWrap**模板是迄今为止读者见到的最典型的使用异常的例子：在**operator[ ]**中使用了一个称作**RangeError**的嵌套类（nested class），如果参数越界，则创建一个**RangeError**类型的异常对象。因为**operator[ ]**的返回值类型是一个引用，所以它不能返回0。（程序中不能有空引用。）这是一个真正的异常情况——在当前语境中，程序不知道该做什么；而且不能返回一个不可能的值。在这个例子中，**RangeError**<sup>①</sup>是非常简单的，它假设类的名字能够表达所有必需的信息。如果认为出错对象的索引也很重要的话，可以在**RangeError**类中添加一个数据成员来容纳这个索引值。

这时，程序的输出为：

```
Cat()
Cat()
Cat()
PWrap constructor
allocating a Dog
~Cat()
~Cat()
~Cat()
PWrap destructor
inside handler
```

程序为**Dog**分配存储空间的时候再一次抛出了异常，但是这一次**Cat**数组中的对象被恰当地清理了，没有出现内存泄漏。

### 1.5.3 auto\_ptr

由于在一个典型的C++程序中动态分配内存是频繁使用的资源，所以C++标准中提供了一个**RAII**封装类，用于封装指向分配的堆内存（heap memory）的指针，这就使得程序能够自动释放这些内存。**auto\_ptr**类模板是在头文件**<memory>**中定义的，它的构造函数接受一个指向类属类型（generic type）的指针（无论在代码中使用什么类）作为参数。**auto\_ptr**类模板还重载了指针运算符**\***和**->**，以便对持有的**auto\_ptr**对象的原始指针进行前面介绍的那些运算。这样，读者就可以像使用原始指针一样使用**auto\_ptr**对象。下面的代码演示了如何使用**auto\_ptr**：

① 注意，在这种情况下最好使用C++标准库中定义的异常类——**std::out\_of\_range**。

```

//: C01:Auto_ptr.cpp
// Illustrates the RAII nature of auto_ptr.
#include <memory>
#include <iostream>
#include <cstdint>
using namespace std;

class TraceHeap {
    int i;
public:
    static void* operator new(size_t siz) {
        void* p = ::operator new(siz);
        cout << "Allocating TraceHeap object on the heap "
              << "at address " << p << endl;
        return p;
    }
    static void operator delete(void* p) {
        cout << "Deleting TraceHeap object at address "
              << p << endl;
        ::operator delete(p);
    }
    TraceHeap(int i) : i(i) {}
    int getVal() const { return i; }
};

int main() {
    auto_ptr<TraceHeap> pMyObject(new TraceHeap(5));
    cout << pMyObject->getVal() << endl; // Prints 5
} ///:~

```

**TraceHeap**类重载了**new**运算符和**delete**运算符，这样，就可以准确地看到在程序运行过程中发生了什么事情。注意，像其他类模板一样，**main()**函数里必须在模板参数中指定所要使用的数据类型。但是这里不能使用**TraceHeap\***——**auto\_ptr**已经知道了要存储指定类型的指针。**main()**函数的第二行证实了**auto\_ptr**的**operator->()**函数间接使用了基本的原始指针。最重要的一点是，尽管程序没有显式地删除该原始指针，但是在栈反解的时候，**pMyObject**对象的析构函数会删除该原始指针，下面程序的输出证实了这一点：

```

Allocating TraceHeap object on the heap at address 8930040
5
Deleting TraceHeap object at address 8930040

```

**auto\_ptr**类模板可以很容易地用于指针数据成员。由于通过值引用的类对象总会被析构，所以当对象被析构时，这个对象的**auto\_ptr**成员总是能释放它所封装的原始指针。<sup>①</sup>

### 1.5.4 函数级的try块

由于构造函数能够抛出异常，读者可能希望处理在对象的成员或其基类子对象被初始化的时候抛出的异常。为了做到这一点，可以把这些子对象的初始化过程放到函数级**try**块中。与通常的语法不同，作为构造函数初始化部分的**try**块是构造函数的函数体，而相关的**catch**块紧跟着构造函数的函数体，就像下面这个例子中所写的一样：

```

//: C01:InitExcept.cpp {-bor}
// Handles exceptions from subobjects.
#include <iostream>
using namespace std;

```

① 有关**auto\_ptr**的详细信息，请参考Herb Sutter在1999年10月发表的文章“Using auto\_ptr Effectively”——《C/C++ Users Journal》，第63~67页。

```

class Base {
    int i;
public:
    class BaseExcept {};
    Base(int i) : i(i) { throw BaseExcept(); }
};
class Derived : public Base {
public:
    class DerivedExcept {
        const char* msg;
    public:
        DerivedExcept(const char* msg) : msg(msg) {}
        const char* what() const { return msg; }
    };
    Derived(int j) try : Base(j) {
        // Constructor body
        cout << "This won't print" << endl;
    } catch(BaseExcept&) {
        throw DerivedExcept("Base subobject threw");
    }
};

int main() {
    try {
        Derived d(3);
    } catch(Derived::DerivedExcept& d) {
        cout << d.what() << endl; // "Base subobject threw"
    }
} ///:~

```

注意，在**Derived**类的构造函数中，初始化列表处在关键字**try**和构造函数的函数体之间。如果在构造函数中发生异常，**Derived**类所包含的对象也就没有构造完成，因此程序返回到创建该对象代码的地方（构造函数的调用者）是没有意义的。由于这个原因，惟一合理的做法就是在函数级的**catch**子句中抛出异常。

尽管不是非常有用，C++还是允许在所有函数中使用函数级**try**块，下面的例子说明了这种用法：

```

//: C01:FunctionTryBlock.cpp {-bor}
// Function-level try blocks.
// {RunByHand} (Don't run automatically by the makefile)
#include <iostream>
using namespace std;

int main() try {
    throw "main";
} catch(const char* msg) {
    cout << msg << endl;
    return 1;
} ///:~

```

在这种情况下，**catch**块中的代码可以像函数体中的代码一样正常返回。这种形式的函数级**try**块与在函数中添加**try-catch**来环绕所有代码没有什么区别。

## 1.6 标准异常

读者也可以使用标准C++库中定义的异常。一般来说，使用标准异常类比用户自己定义异常类要方便快捷得多。如果标准类不能满足要求，也可以把它们作为基类来派生出自己的异常类。

所有的标准异常类归根结底都是从**exception**类派生的，**exception**类的定义在头文件**<exception>**中。**exception**类的两个主要派生类为**logic\_error**和**runtime\_error**，这两个类的定义在头文件**<stdexcept>**中（这个头文件包含 **<exception>**）。**logic\_error**类用于描述程序中出现的逻辑错误，例如传递无效的参数。运行时错误（runtime error）是指那些无法预料的事件所造成的错误，例如硬件故障或内存耗尽。**logic\_error**和**runtime\_error**都提供了一个参数类型为**std::string**的构造函数，这样就可以将消息保存在这两种类型的异常对象中，通过**exception::what()**函数，读者可以从对象中得到它所保存的消息，如下面程序所示：

```
//: C01:StdExcept.cpp
// Derives an exception class from std::runtime_error.
#include <stdexcept>
#include <iostream>
using namespace std;

class MyError : public runtime_error {
public:
    MyError(const string& msg = "") : runtime_error(msg) {}
};

int main() {
    try {
        throw MyError("my message");
    } catch(MyError& x) {
        cout << x.what() << endl;
    }
}
//:~
```

尽管**runtime\_error**的构造函数把消息保存在它的**std::exception**子对象中，但是**std::exception**并没有提供一个参数类型为**std::string**的构造函数。用户最好从**runtime\_error**类或**logic\_error**类（或这两个类中某个类的派生类）来派生自己的异常类，而不要直接从**std::exception**类派生。

下面的几个表格描述了标准异常类：

<b>exception</b>	这个类是由C++标准库为所有抛出异常的类提供的基类。读者可以调用 <b>what()</b> 函数并取得 <b>exception</b> 对象初始化时被设置的可选字符串
<b>logic_error</b>	从 <b>exception</b> 类派生。报告程序逻辑错误，通过检查代码，能够发现这类错误
<b>runtime_error</b>	从 <b>exception</b> 类派生。报告运行时错误，只有在程序运行时，这类错误才可能被检测到

输入输出流异常类**ios::failure**也是从**exception**派生的，但是它没有子类。

读者可以直接使用下面两个表中所列的异常类，或者把它们作为基类来派生自己的更加具体的异常类。

从 <b>logic_error</b> 派生的异常类	
<b>domain_error</b>	报告违反了前置条件
<b>invalid_argument</b>	表明抛出这个异常的函数接收到了一个无效的参数
<b>length_error</b>	表明程序试图产生一个长度大于等于 <b>npos</b> 的对象（ <b>npos</b> 是环境尺寸大小的最大可能值，通常为 <b>std::size_t</b> ）
<b>out_of_range</b>	报告一个参数越界错误

<b>bad_cast</b>	抛出这个异常的原因是在运行时类型识别 (runtime type identification) 中发现程序执行了一个无效的动态类型转换 ( <b>dynamic_cast</b> ) 表达式 (见第8章)
<b>bad_typeid</b>	当表达式 <b>typeid(*p)</b> 中的参数 <b>p</b> 是一个空指针时抛出这个异常。(这也是运行时类型识别的特性, 见第8章)

从runtime_error派生的异常类	
<b>range_error</b>	报告违反了后置条件
<b>overflow_error</b>	报告一个算术溢出错误
<b>bad_alloc</b>	报告一个失败的存储分配

## 1.7 异常规格说明

有时不要求程序提供资料告诉函数的使用者在函数使用时会抛出什么异常。但是, 如果这样做, 函数的使用者就无法确定如何编码来捕获所有可能的异常, 所以这种做法通常被认为是不友好的。如果函数的使用者可以得到源代码, 他们就可以通过查找**throw**语句来找到函数所抛出的异常, 但是, 以库的形式提供的函数通常是不包含源代码的。好的文档能够弥补这一缺陷, 但是有多少软件项目能够提供编写良好的文档呢? C++提供一种语法来告诉使用者函数所抛出的异常, 这样他们就能正确处理这些异常了。这就是可选的异常规格说明 (exception specification), 它是函数声明的修饰符, 写在参数列表的后面。

异常规格说明再次使用了关键字**throw**, 函数可能抛出的所有可能异常的类型应该被写在**throw**之后的括号中。这里的函数声明如下所示:

```
void f() throw(toobig, toosmall, divzero);
```

在涉及异常的情况下, 传统的函数声明:

```
void f();
```

意味着函数可能抛出任何类型的异常。下面的函数声明

```
void f() throw();
```

意味着函数不会抛出任何异常 (最好确认一下, 这个函数所调用的所有函数也不会抛出异常! )。

从好的编码策略、好的文档和便于函数调用这几个方面来说, 当读者编写可能抛出异常的函数时, 最好考虑使用异常规格说明。(在这一章的后面, 将会讨论这一方针的变化。)

### 1. unexpected() 函数

如果函数所抛出的异常没有列在异常规格说明的异常集中, 那将会出现什么情况呢? 在这种情况下, 一个特殊的函数**unexpected()**将会被调用。默认的**unexpected()**函数会调用本章前面所讲到的**terminate()**函数。

### 2. set\_unexpected() 函数

像**terminate()**函数一样, **unexpected()**可以提供一种机制设置自己的函数来响应意外的异常 (unexpected exception)。读者可以调用函数**set\_unexpected()**来完成这件事, 类似于**set\_terminate()**, **set\_unexpected()**函数使用一个函数指针作为参数, 这个指针所指向的函数没有参数, 而且其返回值类型为**void**。因为**set\_unexpected()**函数返回了**unexpected()**函数指针先前的值, 所以可以保存这个值, 并且在以后恢复它。为了要使用**set\_unexpected()**函数, 编程人员必须在代码中包含头文件**<exception>**。下面这个例

子用于显示迄今为止这一部分所讨论内容的简单应用：

```

//: C01:Unexpected.cpp
// Exception specifications & unexpected(),
//{-msc} (Doesn't terminate properly)
#include <exception>
#include <iostream>
using namespace std;

class Up {};
class Fit {};
void g();

void f(int i) throw(Up, Fit) {
    switch(i) {
        case 1: throw Up();
        case 2: throw Fit();
    }
    g();
}

// void g() {}           // Version 1
void g() { throw 47; } // Version 2

void my_unexpected() {
    cout << "unexpected exception thrown" << endl;
    exit(0);
}

int main() {
    set_unexpected(my_unexpected); // (Ignores return value)
    for(int i = 1; i <=3; i++)
        try {
            f(i);
        } catch(Up) {
            cout << "Up caught" << endl;
        } catch(Fit) {
            cout << "Fit caught" << endl;
        }
    } //::~~

```

创建**Up**类和**Fit**类作为异常类。虽然异常类通常都很小，但是可以用它们来保存附加信息提供给异常处理器作为参考。

函数**f()**在其异常规格说明中声明仅会抛出**Up**和**Fit**类型的异常，但是从函数的定义来看却不是这样的。函数**g()**的第1个版本（Version 1）被函数**f()**调用时不会抛出任何异常。但是如果有人修改了函数**g()**，使它抛出一个不同类型的异常（就像这个例子中的函数**g()**的第2个版本（Version 2）抛出一个**int**型异常），那么函数**f()**的异常规格说明就违反了规则。

按照自定义**unexpected()**函数的格式要求，**my\_unexpected()**函数没有参数和返回值。这个函数只是显示一条消息，表明它被调用了，然后退出程序（在这里使用**exit(0)**），这样编写本书时所使用的**make**程序就不会失败了）。新的**unexpected()**函数中不能有**return**语句。

在**main()**函数中，**try**块位于**for**循环的内部，因此，所有的可能情况都被执行了。使用这种方式，程序可以实现类似异常恢复的功能。把**try**块嵌套在**for**、**while**、**do**或**if**块中，并且触发异常来试图解决问题；然后重新测试**try**块中的代码。

仅**Up**和**Fit**异常能够被捕获，因为函数**f()**的编写者称只有这两种异常会被触发。函数**g()**

的第2个版本使得**my\_unexpected()**被调用, 因为**f()**抛出了一个**int**型的异常。

在调用**set\_unexpected()**的时候, 函数的返回值被忽略了, 如果希望在某个时刻恢复先前的**unexpected()**, 读者可以参考本章前面所讲的**set\_terminate()**例子, 将**set\_unexpected()**的返回值保存在一个指向函数的指针中。

典型的**unexpected**处理器会将错误记入日志, 然后调用**exit()**终止程序。它也可以抛出另外一个异常(或重新抛出相同的异常)或调用**abort()**。如果它抛出的异常类型不再违反触发**unexpected**的函数的异常规格说明, 那么程序将恢复到这个函数被调用的位置重新开始异常匹配。(这是**unexpected()**函数特有的行为。)

如果**unexpected**处理器所抛出的异常还是不符合函数的异常规格说明, 下列两种情况之一将会发生:

1) 如果函数的异常规格说明中包括**std::bad\_exception** (在**<exception>**中定义), **unexpected**处理器所抛出的异常会被替换成**std::bad\_exception**对象, 然后, 程序恢复到这个函数被调用的位置重新开始异常匹配。

2) 如果函数的异常规格说明中不包括**std::bad\_exception**, 程序会调用**terminate()**函数。

下面的程序演示了这种行为:

```
//: C01:BadException.cpp {-bor}
#include <exception>    // For std::bad_exception
#include <iostream>
#include <cstdio>
using namespace std;

// Exception classes:
class A {};
class B {};

// terminate() handler
void my_thandler() {
    cout << "terminate called" << endl;
    exit(0);
}

// unexpected() handlers
void my_uhandler1() { throw A(); }
void my_uhandler2() { throw; }

// If we embed this throw statement in f or g,
// the compiler detects the violation and reports
// an error, so we put it in its own function.
void t() { throw B(); }

void f() throw(A) { t(); }
void g() throw(A, bad_exception) { t(); }

int main() {
    set_terminate(my_thandler);
    set_unexpected(my_uhandler1);
    try {
        f();
    } catch(A&) {
        cout << "caught an A from f" << endl;
    }
    set_unexpected(my_uhandler2);
    try {
```



```

    g();
} catch(bad_exception&) {
    cout << "caught a bad_exception from g" << endl;
}
try {
    f();
} catch(...) {
    cout << "This will never print" << endl;
}
} ///:~

```

处理器 **my\_uhandler1()** 抛出一个可以接受的异常(A)，所以程序的执行流程成功地恢复到了第1个 **catch** 块中。处理器 **my\_uhandler2()** 抛出的异常(B)不合法，但是 **g** 的异常规格说明中包括 **bad\_exception**，所以类型为 **B** 的异常被替换成类型为 **bad\_exception** 对象，所以第2个 **catch** 也成功了。由于 **f** 的异常规格说明中不包括 **bad\_exception**，所以程序终止处理器 (terminate handler) **my\_thandler()** 被调用了。程序的输出为：

```

caught an A from f
caught a bad_exception from g
terminate called

```

### 1.7.1 更好的异常规格说明

读者可能会觉得现行的异常规格说明规范不太好，而

```
void f();
```

应该表示函数不会抛出异常。如果程序员想抛出任意类型的异常，他应该写成如下形式：

```
void f() throw(...); // Not in C++
```

这确实是一种改进，因为函数的声明会变得更加明确。遗憾的是，通过阅读代码，读者不一定能够准确地知道函数是否会抛出异常——例如，内存分配失败会触发异常。更坏的情况是：在异常处理机制出现之前编写的函数会发觉，由于它们所调用的函数抛出了异常，所以它们也不经意地抛出了异常（它们可能会链接到新的可抛出异常的版本）。因此，这种不明确的描述被保存了下来：

```
void f();
```

意味着“我可能会抛出异常，也可能不会抛出异常。”为了避免干扰代码的演化，这种不确定性是必需的。如果读者想明确表示函数 **f** 不会抛出任何异常，可以使用空的异常类型列表，如下所示：

```
void f() throw();
```

### 1.7.2 异常规格说明和继承

类中的每个公有函数本质上来说都是类与用户的一种约定。用户传给函数特定的参数，它执行某种处理并且/或者返回结果。同样的约定必须在派生类中保持有效；否则，派生类和基类之间“是一个 (is-a)”的关系就会被违背。由于异常规格说明在逻辑上也是函数声明的一部分，所以在继承层次结构中也必须保持一致。例如，如果基类的一个成员函数声明它只抛出一种类型的异常 **A**，那么派生类中覆盖这个函数的函数不能在异常规格说明列表中添加其他异常。因为如果添加其他异常，就会造成依赖于基类接口的任何程序崩溃。读者可以在派生类函数的异常规格说明中指定较少的异常或指定为不抛出异常，因为这样不需要用户修改任何代码。读者也可以在派生类函数的异常规格说明中指定任何“是一个 (is-a)” **A** 来代替 **A**。举例如下：



```

//: C01:Covariance.cpp {-xo}
// Should cause compile error. {-mwcc}{-msc}
#include <iostream>
using namespace std;

class Base {
public:
    class BaseException {};
    class DerivedException : public BaseException {};
    virtual void f() throw(DerivedException) {
        throw DerivedException();
    }
    virtual void g() throw(BaseException) {
        throw BaseException();
    }
};

class Derived : public Base {
public:
    void f() throw(BaseException) {
        throw BaseException();
    }
    virtual void g() throw(DerivedException) {
        throw DerivedException();
    }
};
//:~

```

由于**Derived::f()**违反了**Base::f()**的异常规格说明，所以编译器将认为**Derived::f()**是错误的（或者至少给出一个警告）。**Derived::g()**的异常规格说明可以被编译器接受，因为**DerivedException**“是一个（is-a）”**BaseException**（没有其他可能性）。读者可以认为**Base/Derived**和**BaseException/DerivedException**是并行的类层次结构；在派生类中，可以用**DerivedException**的返回值来代替指向异常规格说明中的**BaseException**对象的引用。这种行为被称为协变（covariance）（因为两套类同时在各异的继承层次结构上向下变化）。（回顾在第1卷中曾说过：参数类型不能协变——在覆盖虚函数的时候不允许修改函数的签名。）

### 1.7.3 什么时候不使用异常规格说明

如果阅读标准C++库中定义的函数声明，读者会发现没有一个函数使用了异常规格说明。尽管这看起来很奇怪，但是这种看似奇怪的做法是有原因的：标准C++库主要是由模板组成的，无法知道普通的类或函数会做些什么。例如，读者正在开发一个普通的栈模板，并且在**pop**函数中使用异常规格说明，如下所示：

```
T pop() throw(logic_error);
```

由于读者所能预见到的错误只有栈下溢，读者可能认为在异常规格说明中指定一个**logic\_error**或某种恰当的异常类型是安全的。但是类型**T**的拷贝构造函数可能会抛出异常。那么，**unexpected()**会被调用，程序终止。应用系统无法提供可支持异常处理的保证。当无法知道会触发什么异常时，不要使用异常规格说明。这就是为什么模板类，也就是标准C++库的主要组成部分，不使用异常规格说明的原因——它们将其所知道的异常写在文档中，把剩下的事情交给用户来做。异常规格说明主要是为非模板类准备的。

## 1.8 异常安全

在第7章中，将要深入讨论标准C++库中的容器，包括栈容器。读者将会注意到，**pop()**

成员函数的声明如下：

```
void pop();
```

读者可能感觉很奇怪，**pop()**的返回值类型是**void**。它仅仅删除了栈顶元素。为了获得栈顶元素的值，程序员得在调用**pop()**之前调用**top()**。这种做法有一个很重要的原因就是**stack**必须保证异常安全，在标准C++库设计过程中异常安全是一个至关重要的考虑因素。当面对异常的时候有不同级别的异常安全，但是，顾名思义，异常安全中最重要的一点是正确的语义。

假设读者正在使用动态数组实现一个栈（使用**data**作为数组变量名，使用**count**作为整数计数器的变量名），并且试图编写一个带有返回值的**pop()**函数。这个**pop()**函数的代码如下：

```
template<class T> T stack<T>::pop() {
    if(count == 0)
        throw logic_error("stack underflow");
    else
        return data[--count];
}
```

如果为了得到返回值而调用拷贝构造函数，函数却在最后一行抛出一个异常，当该值返回时会发生什么情况呢？因为发生了异常，函数并没有将应该退栈的元素返回，但是**count**已经减1了，所以函数希望得到的栈顶元素丢失了！问题产生的原因是这个函数试图一次做两件事情：（1）返回值，并且（2）改变栈的状态。最好将这两个独立的动作放到两个独立的函数中，这就是标准的**stack**类的做法。（换句话说，遵守内聚设计原则——每个函数只做一件事情。）异常安全代码能够使对象保持状态的一致性而且能够避免资源泄漏。

读者需要仔细编写自定义的赋值操作符。在第1卷的第12章，读者已经看到**operator=**应该遵守下面的模式：

- 1) 确保程序不是给自己赋值。如果是的话，跳到步骤6。（这是一种严格的最优化。）
- 2) 给指针数据成员分配所需的新内存。
- 3) 从原有的内存区向新分配的内存区拷贝数据。
- 4) 释放原有的内存。
- 5) 更新对象的状态，也就是把指向分配新堆内存地址的指针赋值给指针数据成员。
- 6) 返回 **\*this**。

重要的是，直到所有的新增部件都被安全地分配到内存并初始化之前不要修改对象的状态。一个好的技巧是将步骤2和步骤3放到单独的函数中，这个函数常被叫做**clone()**。下面的例子演示了如何在一个拥有两个指针成员（**theString**和**theInts**）的类中使用这一技术：

```
//: C01:SafeAssign.cpp
// An Exception-safe operator=.
#include <iostream>
#include <new>           // For std::bad_alloc
#include <cstring>
#include <cstddef>
using namespace std;

// A class that has two pointer members using the heap
class HasPointers {
    // A Handle class to hold the data
    struct MyData {
        const char* theString;
        const int* theInts;
```

```

    size_t numInts;
    MyData(const char* pString, const int* pInts,
           size_t nInts)
        : theString(pString), theInts(pInts), numInts(nInts) {}
} *theData; // The handle
// Clone and cleanup functions:
static MyData* clone(const char* otherString,
                    const int* otherInts, size_t nInts) {
    char* newChars = new char[strlen(otherString)+1];
    int* newInts;
    try {
        newInts = new int[nInts];
    } catch(bad_alloc&) {
        delete [] newChars;
        throw;
    }
    try {
        // This example uses built-in types, so it won't
        // throw, but for class types it could throw, so we
        // use a try block for illustration. (This is the
        // point of the example!)
        strcpy(newChars, otherString);
        for(size_t i = 0; i < nInts; ++i)
            newInts[i] = otherInts[i];
    } catch(...) {
        delete [] newInts;
        delete [] newChars;
        throw;
    }
    return new MyData(newChars, newInts, nInts);
}
static MyData* clone(const MyData* otherData) {
    return clone(otherData->theString, otherData->theInts,
                otherData->numInts);
}
static void cleanup(const MyData* theData) {
    delete [] theData->theString;
    delete [] theData->theInts;
    delete theData;
}
public:
    HasPointers(const char* someString, const int* someInts,
                size_t numInts) {
        theData = clone(someString, someInts, numInts);
    }
    HasPointers(const HasPointers& source) {
        theData = clone(source.theData);
    }
    HasPointers& operator=(const HasPointers& rhs) {
        if(this != &rhs) {
            MyData* newData = clone(rhs.theData->theString,
                                    rhs.theData->theInts, rhs.theData->numInts);
            cleanup(theData);
            theData = newData;
        }
        return *this;
    }
    ~HasPointers() { cleanup(theData); }
    friend ostream&
    operator<<(ostream& os, const HasPointers& obj) {
        os << obj.theData->theString << ": ";
        for(size_t i = 0; i < obj.theData->numInts; ++i)
            os << obj.theData->theInts[i] << ' ';

```



```

        return os;
    }
};

int main() {
    int someNums[] = { 1, 2, 3, 4 };
    size_t someCount = sizeof someNums / sizeof someNums[0];
    int someMoreNums[] = { 5, 6, 7 };
    size_t someMoreCount =
        sizeof someMoreNums / sizeof someMoreNums[0];
    HasPointers h1("Hello", someNums, someCount);
    HasPointers h2("Goodbye", someMoreNums, someMoreCount);
    cout << h1 << endl; // Hello: 1 2 3 4
    h1 = h2;
    cout << h1 << endl; // Goodbye: 5 6 7
} ///:~

```

为了方便起见，类**HasPointers**使用**MyData**类作为两个指针的句柄。一旦需要分配更多的内存，不论是在构造函数中还是在赋值操作中，程序最终都会调用第一个**clone**函数来完成。如果第一条使用**new**运算符分配内存的语句失败，则会自动抛出一个**bad\_alloc**异常。如果第二条分配内存的语句（为**theInts**分配内存）失败，系统必须清理为**theString**分配的内存——第一个**try**块捕获了**bad\_alloc**异常。第二个**try**块不是至关重要的，因为只是拷贝**int**和指针（不会触发异常），但是当拷贝对象的时候，它们的赋值操作符可能会触发异常，所以应该清理它们。请注意，在这两个异常处理器中，重新抛出了异常。这是因为在这里系统只是进行资源管理工作，函数的使用者仍然需要知道发生了什么错误，所以系统的异常处理机制让异常沿着函数调用动态链向上传播。不会默默地吞没异常的软件库被称做异常中立的（exception neutral）。对于读者来说，始终需要努力写出异常安全且异常中立的软件库。<sup>①</sup>

如果仔细检查上面的代码，就会发现没有一个**delete**操作会抛出异常。这段代码正是基于这一事实。回忆一下，当程序中用**delete**删除一个对象的时候，这个对象的析构函数会被调用。结果是：事实上，只能假设析构函数不抛出异常，否则无法设计异常安全的代码。不要让析构函数抛出异常。（在本章结束之前将对此进行多次提醒。）<sup>②</sup>

## 1.9 在编程中使用异常

对大多数程序员，尤其是C程序员来说，他们目前使用的程序设计语言不支持异常，所以需要做一些调整。下面是一些在程序设计中使用异常的指导原则。

### 1.9.1 什么时候避免异常

异常并不能解决所有问题；过度使用会造成麻烦。本文下面的部分指出了在哪种情况下不应该使用异常。有关何时应该使用异常的最好建议是：只有当函数不符合它的规格说明时才抛出异常。

#### 1. 不要在异步事件中使用异常

标准C的**signal()**系统及类似系统负责处理异步事件：这些事件是发生在程序流程之外的，

① 如果读者对更深入地分析异常安全问题感兴趣，权威的参考书是Herb Sutter的《Exceptional C++》，Addison-Wesley, 2000。

② 在栈反解过程中，库函数**uncaught\_exception()**返回**true**。因此从技术上来讲，用户可以使用**uncaught\_exception()**来测试当前状态，如果返回**false**，那么就可以让析构函数抛出异常。我们从未见过使用这种技术实现优秀设计的先例，所以只是在脚注中提及。

而且这些事件的发生是程序无法预料的。由于异常和它的处理器必须处在相同的函数调用栈上,所以无法使用C++中的异常机制来处理异步事件。也就是说,异常依赖于程序运行栈上的动态函数调用链(他们有“动态作用域(dynamic scope)”),然而异步事件必须由完全独立的代码来处理,这些代码不是正常程序流程的一部分(典型的例子是:中断服务例程和事件循环)。不要在中断处理程序中抛出异常。

这并不是说异步事件不能与异常发生联系。但是,中断处理程序应该尽快完成工作并返回。处理这种情况的典型方式是,中断处理程序设置一个标记,程序的主干代码同步地检查这个标记。

## 2. 不要在处理简单错误的时候使用异常

如果能得到足够的信息来处理错误,那么就不要再使用异常。程序员应该在当前语境中处理这个错误,而不是将一个异常抛出到更大的(上一层)语境中。

此外,C++在遇到机器层事件如除零错误<sup>①</sup>时不会抛出异常。读者可以认为其他一些机制,如操作系统或硬件会处理这种事件。这样,C++的异常机制可以相当有效,它们被隔离起来只用于处理程序级的异常状况。

## 3. 不要将异常用于程序的流程控制

异常看起来有点像函数返回机制的代替品,也有点像switch语句,因此,读者可能觉得用异常来代替这些普通的语言机制很有吸引力。这是一种错误的想法,部分原因是异常处理系统的效率比普通的程序流控制差很多。异常仅仅是一个非常事件,使用异常要付出一定的代价。同样,如果把异常用于处理错误之外的其他地方,也会令类或函数的使用者带来混乱。

## 4. 不要强迫自己使用异常

某些程序是相当简单的(例如一些小型的实用程序)。程序中可能只需要接收输入数据,进行某些处理。在这些程序中,可能在分配内存时失败,打开文件时失败等。遇到这类情况,显示一个消息然后退出程序就可以了,最好把清理工作交给操作系统来处理,而不必费劲地捕获所有异常并释放资源。简单地说,如果读者不需要异常,就不要强迫自己使用它们。

## 5. 新异常,老代码

另一个问题出现在需要对现有的没有使用异常的程序进行修改的情况下。在程序设计中,可能引入了一个使用异常机制的库,并且想知道是否应该修改程序中所有的代码。假设在程序中已经拥有了一个令人满意的错误处理模式,最直接的方法是把使用新库的覆盖范围最大的代码段(可能是main()函数中的所有语句)放到try块中,追加一个catch(...),然后是基本的错误信息。可以进一步精练它们,根据需要的程度,添加更明确的异常处理器来改进这种做法,但是无论如何,新添加的代码应该尽可能的少。更好的方法是把产生异常的代码隔离在try块中,并且编写异常处理器把异常转换成与现有错误处理模式兼容的形式。

当一个编程人员正在编写一个供其他人使用的库时,特别是当无法知道他们如何响应致命性错误条件的时候,慎重地考虑异常非常重要(回忆一下之前对异常安全的讨论,为什么标准C++库中没有使用异常规格说明)。

### 1.9.2 异常的典型应用

在下列情况下请使用异常:

- 修正错误并且重新调试产生异常的函数。
- 在重新调试中的函数外面补偿一些行为以便使程序得以继续执行。

① 某些编译器在这种情况下会抛出异常,但是它们通常提供编译器选项来禁止这种(不常见的)行为。

- 在当前语境中做尽可能多的事情，并把同样类型的异常重新抛出到更高层的语境中。
- 在当前语境中做尽可能多的事情，并将一个不同类型的异常抛出到更高层的语境中。
- 终止程序。
- 将使用普通错误处理模式的函数(尤其是C库函数)封装起来，以使用异常来代替原有的错误处理模式。
- 简化。如果建立的错误处理模式使事情变得更复杂并且难以使用，那么异常可以使错误处理更加简单有效得多。
- 使建立的库和程序更安全。使用异常既是一种短期投资(为了调试方便)也是一种长期投资(为了应用系统的健壮性)。

#### 1. 什么时候使用异常规格说明

异常规格说明就像函数原型：它提醒使用者来编写异常处理代码以及处理什么异常。它提醒编译器这个函数可能抛出异常，让编译器能够在运行时检测违反该异常规格说明的情况。

一个程序设计人员不能总是通过检查代码来预测某个特定的函数会抛出什么异常。有时候函数会产生无法预料的异常，有时候一个不抛出异常的旧函数会被一个抛出异常的新函数替换掉，并且迫使程序调用**unexpected()**。任何时候如果要使用异常规格说明，或调用使用异常规格说明的函数，最好编写自己的**unexpected()**函数，在这个**unexpected()**函数中将消息记入日志，然后抛出异常或终止程序。

如前所述，应该避免在模板类中使用异常规格说明，因为无法预料模板参数类(template parameter classes)所抛出的异常的类型。

#### 2. 从标准异常开始

在编写自己的异常类之前检查标准C++库中所定义的异常类。如果标准异常类符合系统设计的要求，则可能会使用户更容易理解和处理。

如果标准库中没有定义用户所需的异常类，应尽量从现有的标准异常类中继承出一个。如果用户能够使用**exception()**类中定义的接口函数**what()**，那么用户的异常类将显得非常友好。

#### 3. 嵌套用户自己的异常

如果为用户自己的特定类创建异常类，最好在这个特定类中或包含这个特定类的名字空间中嵌套异常类，这就为读者提供了一个明确的信息——这个异常类仅在用户自己的特定类中使用。另外，这也避免了污染全局名字空间。

即使用户自己的异常类是从C++标准异常类中派生的，用户也可以嵌套它们。

#### 4. 使用异常层次结构

异常层次结构为用户的类或库可能遇到的不同类型的重要错误提供了一个有价值的分类方法。这种方法给用户提供了有价值的信息，帮助他们组织代码，使他们能够有选择地忽略所有异常的附加的类型而仅仅捕获基类类型。另外，后来添加到异常类层次结构中的从相同基类继承的任何异常不会迫使用户重写现有的代码——针对基类的异常处理器将会捕获到这个新异常。

标准C++异常类是异常层次结构的一个好的范例。如果可以的话，应基于这些异常类来创建用户自己的异常类。

#### 5. 多重继承(MI)

读者在研读第9章的时候就会发现，惟一必须用到多重继承的情况是：当需要将一个对象指针向上类型转换成两个不同的基类类型时——也就是说，读者同时需要这两个基类的多态行为。异常层次结构在这种情况下也是有用的，因为多重继承异常类的任何一个基类的异常处理器都能够处理这个异常。

## 6. 通过引用而不是通过值来捕获异常

正如读者在“异常匹配”那节内容所见到的，应该通过引用来捕获异常，这么做有两个原因：

- 当异常对象被传递到异常处理器中的时候，避免进行不必要的对象拷贝。
- 当派生类对象被当做基类对象捕获时，避免对象切割。

尽管可以抛出并且捕获指针类型的异常，但是如果这么做的话，将会在代码中引入紧耦合——抛出异常的代码和捕获异常的代码，必须就如何为异常对象分配内存和如何清理异常对象达成一致。由于在堆耗尽的时候也可能会触发异常，所以这也造成了一个问题。如果程序抛出异常对象，异常处理系统负责处理所有与存储有关的问题。

## 7. 在构造函数中抛出异常

由于构造函数没有返回值，有两种方法来报告在构造对象期间发生的错误。

- 设置一个非局部的标记，并且希望用户检查它。
- 返回一个未完成的创建对象，并且希望用户检查它。

这个问题至关重要，因为C程序员希望所有的对象创建工作总是成功的，这一点在C语言中并不是不合理的，因为C语言中的类型非常简单。但是在C++程序中，不理睬构造函数中出现的故障而继续运行，肯定会导致灾难性的后果，所以构造函数是抛出异常最重要的位置之一——现在用户有了一种安全有效的方式来处理构造函数异常。然而，当构造函数抛出异常时，用户必须注意对象内部的指针和它的清理方式。

## 8. 不要在析构函数内部触发异常

因为析构函数会在抛出其他异常的过程中被调用，所以绝不要在析构函数中抛出异常或在析构函数中执行其他可能触发抛出异常的操作。如果在析构函数中抛出异常，这个新的异常可能会在现存的异常（其他异常）到达`catch`子句之前被抛出，这会导致程序调用`terminate()`函数。

如果在析构函数中调用的函数可能会抛出异常，应该在这个析构函数中编写一个`try`块，并把这些函数调用放到`try`块中，析构函数必须自己处理所有这些异常。绝对不能有任何一个异常从析构函数中抛出。

## 9. 避免悬挂指针

请看这一章前面的`Wrapped.cpp`程序。如果需要给指针分配资源，那么悬挂指针通常意味着构造函数的弱点。如果在构造函数中抛出异常，因为指针没有析构函数，那么这些资源将无法释放。请使用`auto_ptr`或其他智能指针（smart pointer）类型<sup>①</sup>来处理指向堆内存的指针。

## 1.10 使用异常造成的开销

当异常被抛出时，将造成相当多的运行时开销（但是，这是有益的开销，因为对象被自动清理了！）。由于这种原因，不要将异常作为正常控制流的一部分使用，无论这种想法看起来多么精巧诱人。异常应该很少发生，所以开销主要是由异常造成的而不是由正常执行的代码造成的。异常处理机制的重要设计目标之一是，当异常没有发生时，它不应该影响系统的运行速度；也就是说，如果不抛出异常，那么代码的运行速度就像没有使用异常处理机制时一样快。这么说是否正确，依赖于用户所使用的特定编译器的实现方式。（参考这一节的后面部分对“零代价模型（zero-cost model）”的描述。）

① 在网址[http://www.boost.org/libs/smart\\_ptr/index.htm](http://www.boost.org/libs/smart_ptr/index.htm)可以找到增强的智能指针类型。下一版的标准C++正在考虑包含这些智能指针类型中的一部分。



可以这样认为，一个**throw**表达式就像是一个特殊的系统函数调用，它接收异常对象作为参数并且沿着执行调用链向上回溯。为了完成这项工作，编译器需要在栈上放置额外的信息，来辅助栈反解过程。为了理解这些内容，用户需要了解有关运行栈（runtime stack）的知识。

每当函数被调用的时候，有关这个函数的信息被压到运行栈顶部的活动记录实例（activation record instance, ARI）中，也叫栈结构（stack frame）。典型的栈结构包含调用函数的指令所在的地址（这样，程序的执行流程可以返回到这个地址），指向这个函数静态父对象的ARI（某个作用域，它在词法上包含被调用函数，这样这个函数就可以访问全局变量了）的指针，和指向调用函数的指针（它的动态父函数）。沿着动态父函数链反复追踪所得到的逻辑结果路径就是动态链，或称其为调用链，读者在这一章的前面见到过它。这就是为什么当异常抛出时执行流程能够回溯，这种机制使得在彼此缺乏了解的情况下开发出来的程序的各个部分，能够在运行时互相传递出错信息。

对于异常处理机制系统允许栈反解，每个函数额外的异常相关信息，必须对每一个栈结构来说都是可用的。这些信息描述了哪个析构函数应该被调用（因此，局部对象可以被清理），这些信息显示了当前函数是否有**try**块，而且这些信息列出了与**try**块相关的**catch**子句能够捕获哪些异常。这些额外信息会造成存储空间消耗，所以支持异常处理机制的程序要比不支持异常处理机制的程序大<sup>①</sup>。因为在运行期间生成扩展栈结构的逻辑必须由编译器生成，所以使用异常处理的程序在编译时也较大。

为了演示这一点，在这里使用Borland C++ Builder和Microsoft Visual C++<sup>②</sup>：分别在支持异常处理机制和不支持异常处理机制的模式下编译下面的程序：

```

//: C01:HasDestructor.cpp {0}
class HasDestructor {
public:
    ~HasDestructor() {}
};

void g(); // For all we know, g may throw.

void f() {
    HasDestructor h;
    g();
} ///:~

```

如果允许异常处理，编译器必须为**f()**保存有关析构函数**~HasDestructor()**在运行时的大量信息到**ARI**（活动记录实例）中（这样即使**g()**抛出异常，**f()**也能正确地销毁对象**h**）。下表总结了编译结果文件（.obj）的大小（单位：字节）。

编译器\模式	支持异常处理	不支持异常处理
Borland	616	234
Microsoft	1162	680

不要把两种模式之间文件大小的百分比看得太重。请记住，典型情况下异常（应该）只构成程序的很小一部分，其空间开销是相当小的（通常只占总开销的5%~15%）。

① 这取决于在不使用异常的情况下用户必须插入多少代码来检查返回值。

② Borland在默认情况下允许异常处理；使用**-x**编译器选项来禁止异常处理。Microsoft在默认情况下不允许异常处理；使用**-GX**选项开启异常处理。两种编译器都使用**-c**选项作为只执行编译过程的选项。



额外的管理工作会降低执行速度，但是聪明的编译器会避免这种情况。由于与异常处理代码和局部对象偏移量有关的信息只在编译时刻计算一次，这些信息可以保存在与每个函数相关的单独位置中，而不是保存在每个ARI中。我们基本上已经从每个活动记录实例中消除了异常的空间开销，因此也避免了压栈操作造成的附加时间开销。这种方法称为异常处理的零代价（zero-cost）模型<sup>①</sup>，早先提及的优化存储被认为是影子栈（shadow stack）。<sup>②</sup>

## 1.11 小结

错误恢复是程序员在编写每个程序时最关心的内容。当使用C++创建程序的组件供他人使用时，错误恢复是非常重要的。要创建一个健壮的系统，那么它的每一个组件都必须足够健壮。

C++中异常处理的目标是简化创建庞大、可靠程序所需的工作，用更少的代码使软件开发者对程序中是否仍然包含未处理的错误拥有更多信心。在不损失或损失很少性能的情况下，在很少干扰现存代码的情况下，现在实现了这一目标。

基本的异常不难掌握；一旦能够掌握，就可以在程序中开始使用它们。异常是能够为用户要开发的项目提供直接和重大利益的特性之一。

## 1.12 练习

- 1-1 编写三个函数：一个通过返回错误值来指出错误情况，一个设置**errno**标志，最后一个使用**signal()**。编写代码调用这些函数并响应产生的错误。编写第4个函数，这个函数抛出异常。调用这个函数并捕获异常。描述这4种方法的区别，为什么说异常处理机制是一种更好的方法。
- 1-2 创建一个类，这个类含有抛出异常的成员函数。在这个类中嵌套一个类作为异常对象的类型。这个异常类使用一个**const char\***作为参数；这个参数代表一个描述字符串。创建一个抛出这种异常的成员函数。（在函数的异常规格说明中描述这种异常。）编写一个**try**块调用这个成员函数，写一个**catch**子句通过显示描述字符串的方式处理这个异常。
- 1-3 重新编写第1卷第13章中的**Stash**类，为**operator[]**抛出**out\_of\_range**异常。
- 1-4 编写一个普通的**main()**函数，捕获所有的异常并报告错误。
- 1-5 创建一个类，这个类带有自己的**new**运算符。这个运算符为十个对象分配内存，并且在为第11个对象分配内存时抛出“run out of memory”（内存用完）异常。并且添加一个静态成员函数用于回收这个内存。创建**main()**函数，其中包含**try**块和**catch**子句，在**catch**子句中调用回收内存的子例程。把这些代码放在一个**while**循环中，用于演示从异常恢复并继续执行的过程。
- 1-6 创建一个抛出异常的析构函数，编写代码来为自己证明这是一个坏主意。（在能够捕获现有异常的异常处理器被调用之前，如果一个新的异常被抛出，会调用**terminate()**。）
- 1-7 证明所有异常对象（被抛出的异常对象）都会被正确销毁。
- 1-8 证明如果我们在堆上创建一个异常对象，并且抛出指向这个对象的指针，那么这个对象不会被清理。

① GNU C++编译器默认使用零代价模型。Metrowerks Code Warrior for C++也有一个选项，能够选择使用零代价模型。

② 感谢Scott Meyers和Josee Lajoie在零代价模型上的洞察力。读者可以在Josee的精彩文章“Exception Handling: Behind the Scenes,” C++ Gems, SIGS, 1996中找到有关异常如何工作的更多信息。

- 1-9 编写一个带有异常规格说明的函数，这个函数能够抛出4种异常：一个**char**、一个**int**、一个**bool**和一个自己的异常类。在**main()**函数中捕获每种异常，并且验证这些捕获的异常。从标准异常类派生自己的异常类。使用下述方法编写**main()**函数：系统从异常中恢复并尝试重新执行抛出异常的函数。
- 1-10 修改上一个练习，让函数抛出一个违反异常规格说明的**double**类型的异常。在自己的**unexpected**处理函数中捕获这个违反异常规格说明的错误，显示一个消息然后优雅地退出程序（也就是说不要调用**abort()**）。
- 1-11 编写一个**Garage**类，这个类包含一个**Car**，这个**Car**的**Motor**出了故障。在**Garage**类的构造函数中使用函数级**try**块用于捕获**Car**对象初始化时抛出的异常（从**Motor**类抛出的异常）。从**Garage**类构造函数的异常处理器中抛出一个不同的异常，并在**main()**函数中捕获这个异常。



## 防御性编程

编写“完美的软件”对开发者来说可能是一个难以达到的目标，但是应用一些常规的防御性技术，对于提高代码的质量将会大有帮助。

尽管典型的软件产品的复杂性保证了测试人员总有做不完的工作，然而，程序设计人员仍然渴望创造零缺陷的软件。面向对象设计技术为开发大型项目解决了很多困难，但是最终用户还得自己编写循环和函数。这些“细微处编程”(programming in the small)的详细内容成为用户设计的较大组件所需的构建块。如果循环偶尔突然退出，或者函数只是在“大多数”时候能够计算出正确的结果，那么，不管系统的整体结构（方法论）是多么的优秀，最终还是会陷入麻烦之中。本章中读者将会学习到一些经验，不管项目是什么规模的，这些经验都能够帮助程序员创建出健壮的代码。

代码的其中一个内涵是对问题解决方法的描述。在设计循环的时候，程序员应该能够清楚地告诉读者（包括程序员自己）其准确的想法是什么。在程序中某个特定的地方，应该能够大胆地声明某些条件或其他一些控制方法。（如果不能做出这种声明，只能说明实际上还未解决这个问题。）这种声明称为不变量（invariant），因为在代码中它们出现的那一点上它们应该恒为真；否则，要么是设计有缺陷，要么是代码没有正确地反映程序设计人员的设计意图。

考虑这样一个实现Hi-Lo猜谜游戏的程序。某个人在1到100之间任选一个数，另一个人猜这个数。（现在我们让计算机猜这个数。）选数的人告诉猜数的人他猜的数比正确的数大，还是比正确的数小或是正好相等。对猜数的人来说，最好的策略是进行二分查找，选择待查找数字范围的中间点。根据选数的人回答的“大”或“小”，猜数的人能够知道这个数到底在该范围的哪一半中。重复这个过程，每次重复都能够把范围缩小一半。那么怎么样编写这个循环来正确模拟猜数过程呢？像下面这样写是不够的：

```
bool guessed = false;
while(!guessed) {
    ...
}
```

因为恶意的用户可能会欺骗编程者，使他们花整天的时间进行猜测。然而每次猜测时能做什么样的假设呢？换句话说，在每个循环中设计什么样的条件来控制循环的迭代次数呢？

现假定：秘密的数字是在当前有效的未猜过的数的范围之内：[1, 100]。假设用**low**和**high**两个变量标记数字范围的端点。每次进入循环，在循环开始的时候，需要确定该秘密的数字在范围[**low**, **high**]之内，每次迭代结束之后重新计算数的范围，使其在进行下一次循环迭代时仍然含有该秘密数字。

这样做的目标是在编写代码的时候表示出循环的不变量条件，使得程序可以在运行的时候检测到违背条件的情况。遗憾的是，由于计算机不知道秘密的数字，程序员不能在代码中直接表达这个条件，但是至少能够写一段这种效果的注释：

```
while(!guessed) {
    // INVARIANT: the number is in the range [low, high]
    ...
}
```

当用户回答猜测结果太大或太小（即超出秘密数字的可能范围）时，会出现什么情况呢？这样的欺骗会造成新计算出来的子范围不包括秘密数字。因为一个谎言总会导致另一个谎言，最终会使猜数范围缩减到不包含任何数字（由于每次将猜数范围缩减一半，而且秘密数字并不在范围内）。下面的程序可以表示这种情况。

```

//: C02:HiLo.cpp {RunByHand}
// Plays the game of Hi-Lo to illustrate a loop invariant.
#include <cstdlib>
#include <iostream>
#include <string>
using namespace std;

int main() {
    cout << "Think of a number between 1 and 100" << endl
         << "I will make a guess; "
         << "tell me if I'm (H)igh or (L)ow" << endl;
    int low = 1, high = 100;
    bool guessed = false;
    while(!guessed) {
        // Invariant: the number is in the range [low, high]
        if(low > high) { // Invariant violation
            cout << "You cheated! I quit" << endl;
            return EXIT_FAILURE;
        }
        int guess = (low + high) / 2;
        cout << "My guess is " << guess << ". ";
        cout << "(H)igh, (L)ow, or (E)qual? ";
        string response;
        cin >> response;
        switch(toupper(response[0])) {
            case 'H':
                high = guess - 1;
                break;
            case 'L':
                low = guess + 1;
                break;
            case 'E':
                guessed = true;
                break;
            default:
                cout << "Invalid response" << endl;
                continue;
        }
    }
    cout << "I got it!" << endl;
    return EXIT_SUCCESS;
} //:~

```

条件表达式**if(low > high)**可以发现违反不变量条件的情况，因为如果用户总是说实话，那么在用完这些猜测前总能找到这个秘密数字。

也可以使用标准C的技术通过从**main()**函数中返回不同的值来向调用者报告程序的状态。用语句**return 0**的可携带值来表示程序执行成功，但是没有可携带的值可作为表示程序执行失败的返回值。因此，可以在这种情况下使用**<cstdlib>**中声明的宏：**EXIT\_FAILURE**表示程序执行失败的返回值。为了代码的一致性，无论何时使用**EXIT\_FAILURE**，我们也使用**EXIT\_SUCCESS**来表示程序执行成功，虽然**EXIT\_SUCCESS**总是被定义成0。

## 2.1 断言

在Hi-Lo程序执行依赖于用户输入的情况下，无法防止在程序运行过程中出现违反不变量条件的事件发生。然而，不变量条件通常仅仅依赖于编写的代码，所以这些不变量条件始终持有程序设计是否已经正确实现的证据。在这种情况下，可以明确地使用断言（assertion），断言是一个肯定的语句，（只要能证明在程序的执行过程中断言恒真，就证明了程序的正确性。）用来肯定显示设计意图。

假设现在正在实现一个整数向量（vector）：一个可以按照需求扩展的数组。添加一个元素到向量中的函数必须首先检查在数组的下面的位置是否有空闲的单元；如果没有空闲单元，函数必须请求更多的堆空间，而将现有的元素拷贝到新分配的内存空间中，最后把这个新元素添加到数组中（并且释放旧的数组）。如下所示：

```
void MyVector::push_back(int x) {
    if(nextSlot == capacity)
        grow();
    assert(nextSlot < capacity);
    data[nextSlot++] = x;
}
```

在这个例子中，**data**是一个整型的动态数组，有**capacity**个单元，前**nextSlot**个单元已经被使用了。**grow()**函数的作用是扩大**data**数组，使新数组的**capacity**值比**nextSlot**大。**MyVector**的行为是否正确依赖于设计决定，如果其他支持代码正确，**MyVector**就不会出错。可以使用定义在头文件<cassert>中的**assert()**宏断言这种情况。

标准C库中的**assert()**宏是简明扼要的并且也可携带返回信息。如果参数赋值得到的条件为非零值，程序将不受干扰地继续运行；否则，引发断言错误的表达式和其所所在的源文件的文件名、这个断言所在行的行号都会被一起送到标准错误输出信道打印出来，然后程序异常终止。这种反应方式太过激烈吗？实际上，当基本设计中的假定已经失败时，让程序继续执行会造成更加剧烈的反应。如果出现了这种情况，就应该修改程序。

如果所有的事情都进行得很好，则应该在配置最终产品之前完整地测试代码，在测试过程中不应该出现触发断言错误的情况。（本章随后会讲更多有关测试的问题。）视应用程序的性质而定，运行时检测所有断言所耗费的机器周期可能会大大降低程序的执行效率，以至严重影响这个系统在该领域的应用。如果是这样的话，定义**NDEBUG**宏并重新编译程序，会自动去掉所有断言代码。

现在来看看断言是如何工作的吧，注意，典型的**assert()**实现如下所示：

```
#ifdef NDEBUG
#define assert(cond) ((void)0)
#else
void assertImpl(const char*, const char*, long);
#define assert(cond) \
    ((cond) ? (void)0 : assertImpl(???))
#endif
```

当定义了**NDEBUG**宏的时候，这段代码蜕化成表达式**(void) 0**，再加上写在每个**assert()**后面的分号，所以最终留在编译器流中的内容仅仅是一条无意义的语句。如果没有定义**NDEBUG**宏，**assert(cond)**被扩展成条件语句，当**cond**的值为零时，调用与编译器相关的函数（称为**assertImpl()**），调用这个函数时需要三个参数，这三个参数分别是：断言语句所在文件的文件名、断言语句所在行的行号和一个字符串，这个字符串表示**cond**的文本

形式。(在这个例子中,使用“???”来代替这些参数,上面提到的字符串文本是在这里确定的,断言语句所在文件的文件名和断言语句所在行的行号也是在文件中宏出现的位置确定的。如何得到这些值对于这个问题的讨论来说并不重要。)如果想开启或关闭程序中某些位置的断言,不但必须包含**#define**或**#undef NDEBUG**,而且必须重新包含**<cassert>**。当预处理器遇见它们时对宏进行赋值,并且无论**NDEBUG**是什么状态都将其应用在包含的位置上。最常用的定义**NDEBUG**的方式是作为编译器选项给整个程序定义,不管是通过可视化开发环境的项目设置还是通过命令行,如下所示:

```
mycc -DNDEBUG myfile.cpp
```

大多数编译器使用**-D**标记来定义宏的名字。(为上面的编译器**mycc**替换要编译的可执行文件的文件名。)这种方法的好处是,可以把断言留在源文件中作为不可多得的珍贵文档使用,而不会在运行时造成性能损失。因为当定义了**NDEBUG**,断言中的代码就会消失,所以确保不在断言中做额外操作是至关重要的。断言中只能包含不会修改程序状态的测试条件。

是否应该在发行版中使用**NDEBUG**仍然有争论。Tony Hoare是最有影响的计算机科学家之一,<sup>①</sup>他比喻说,关掉类似断言的运行时检查,就像一个热衷于航海的人,当他在陆地上训练的时候穿着救生衣,然而当他下海的时候却脱掉了救生衣。<sup>②</sup>断言在软件产品中失灵所造成的问题远比效率降低要严重得多,因此要做出明智的选择。

不是所有情况下都应该使用断言。如第1章所示,用户造成的错误和运行时资源故障应该用抛出异常以信号的方式来通知系统。读者可能希望当粗略描述代码的时候,在大多数错误情况下使用断言,并决心在随后的编码过程中用健壮的异常处理来代替它们。这是一种很诱人的想法。由于在随后的修改过程中,可能会漏掉某些断言,所以,像对待其他的诱惑一样,必须要十分小心。记住:断言的意图是验证设计决定,造成它失败的惟一原因应该是程序逻辑有缺陷。理想的结果是在开发阶段就解决掉所有违背断言的情况。如果某个条件不完全在程序的控制之下,那么不要对这个条件使用断言(例如,依赖于用户输入的条件)。特别是不应该使用断言来验证函数的参数;在参数错误的情况下,应该抛出**logic\_error**异常。

用断言作为工具来确保程序的正确性是Bertrand Meyer在其所著的《Design by Contract methodology》书中正式提出来的。<sup>③</sup>每一个函数都有一个隐含的与客户程序的约定,给定某一个前置条件,保证会出现某一个后置条件。换句话说,前置条件是使用该函数的必要条件,例如提供某一范围内的参数,后置条件是该函数提供的结果,通过返回值或通过副作用给出。

当客户程序给出了一个无效的输入,必须告诉这些客户程序,它们违反了约定。这并不是终止程序的最好时机(尽管这样做是正当的,因为它们违反了约定),在这种情况下,应该抛出异常。这就是为什么标准C++库有从**logic\_error**类派生的异常类,例如**out\_of\_range**异常类,用以抛出异常。<sup>④</sup>如果这些函数只被程序设计人员自己调用,例如自己设计的类中的私有函数,因为能够控制整体情况并且希望在发行代码之前进行调试,所以使用**assert()**宏是适当的。

后置条件的失败表明程序中有错误,在任何时间使用断言来测试任何不变量都是适当的,

① 他发明了快速排序算法和其他一些东西。

② 引用自“Programming Language Pragmatics”, Michael L. Scott, Morgan-Kaufmann, 2000。

③ 参考他所著的书籍《Object-Oriented Software Construction》, Prentice-Hall, 1994。

④ 这在概念上仍然是一种断言,但是由于不想终止程序的运行,使用**assert()**宏并不恰当。例如,Java 1.4在断言失败的时候抛出异常。

包括在一个函数结束的时候测试后置条件。将这种方法应用于维护对象状态的类成员函数中特别合适。在先前提到的**MyVector**例子中，对于所有的公有成员函数来说合理的不变量应该是：

```
assert(0 <= nextSlot && nextSlot <= capacity);
```

或者，如果**nextSlot**是一个无符号整数，简化的结果是

```
assert(nextSlot <= capacity);
```

这样的不变量称为类不变量 (class invariant)，可以使用断言对它进行适度的强制。对于基类来说，它们的子类扮演了一个分包人的角色，因为它们必须维持基类与其客户之间最初的约定。因此，派生类的前置条件不能超过基类与其客户的约定而再强加额外的要求，并且派生类的后置条件必须至少与基类的后置条件一样多。<sup>①</sup>

确认返回给客户的结果是否正确与测试没有什么不同，所以在这种情况下使用后置条件断言 (post-condition assertions) 就与测试工作重复了。当然，这是一种好的文档，但是不止一个软件开发者被这种用法愚弄了，他们错误地把后置条件断言当成单元测试的一种替代品。

## 2.2 一个简单的单元测试框架

为编写软件而做的所有工作都是为了满足客户需求。<sup>②</sup> 确定这些需求非常困难，它们每天都可能在变化；软件开发人员可能在每周召开的例行项目会议中发现，自己花费一周时间所做的工作并不是用户真正想要的。

如果得不到一个可供参照的系统，然后在它的基础上提出改进意见，人们无法清晰地说明软件需求。最好应该确定一个小的系统，设计、编写、测试这个小系统。在提出改进意见之后，再重新完成它。以迭代方式开发程序的能力是面向对象方法的最大优势之一，但是这需要有才干的程序员，这些程序员应该能够精心制作扩展力非常强的代码。修改现有程序是困难的。

另外一个修改程序的动力来自程序设计人员自己。技艺超群的程序员频繁地改进代码的设计。其实，那些软件行业的旗舰公司推出的被改得乱七八糟、错综复杂的产品，有哪件不被产品维护程序员一再诅咒为费解而又难以修改的拼凑物呢？由于经营成本方面的考虑，迫使编程人员损害系统的功能性，放弃了代码的可扩展性，而这正是保证代码持久性所需要的。“如果程序没有坏掉，就不要修改它”，最终让路于“没法修改它——重写算了。”这种状况必须改变。

幸运的是，现在软件行业越来越倾向于代码重构了，重构是通过改造系统内部的代码从而改进程序的设计，并且不改变程序的行为。<sup>③</sup> 这种改善包括从某个函数中摘录出一个新函数，或者相反，组合几个成员函数为一个成员函数；用某个对象替换一个成员函数；参数化一个成员函数或类；有条件地替换多态性。代码重构有助于代码的进化。

不管修改程序的动力来自于用户还是来自于程序员，今天的修改可能破坏昨天的工作。需要有一种方式来构造代码，随着时间的流逝，这些代码应该能够经得起变化和改进行所带来的负面效应。

① 有一个比较好的短语能帮忙记住这种现象：“不要只索取不付出 (Require no more; promise no less)”，这个短语首先在《C++ FAQs》中被Marshall Cline和Greg Lomow (Addison-Wesley, 1994)创造出来。由于前置条件在派生类中被弱化了，所以称其为逆变的 (contravariant)，相反，后置条件是协变的 (covariant) (这就解释了为什么我们在第1章中提到了异常规格说明的协变)。

② 这一部分基于Chuck的文章，“The Simplest Automated Unit Test Framework That Could Possibly Work” C/C++ Users Journal, Sept. 2000。

③ 关于这个主题的一本好书是Martin Fowler的《Refactoring: Improving the Design of Existing Code》(Addison-Wesley, 2000)。参见<http://www.refactoring.com>。重构在极限编程中是一种至关重要的实践。



极限编程 (extreme programming, XP)<sup>①</sup> 仅仅是众多支持快速开发实践方法中的一种。在这一节中, 要探究一种便于使用的自动单元测试工具框架, 它能够使我们成功地开发出灵活的可扩展的程序。(注意: 测试工程师是软件专业人员, 以测试其他人编写的代码为生, 在软件开发活动中, 这些人更是必不可少的。在这里只不过想描述一种方法, 这种方法能够帮助软件开发人员开发出更好的代码。)

通过编写单元测试程序, 开发者能够对下面的两点关键内容获得足够的信心:

- 1) 我理解需求。
- 2) 我的代码符合需求 (以我所学的知识来说, 它们是最好的)。

先编写单元测试程序是一种能够确保将要编写的代码能够正确工作的最好方法。这种简单的工作能够帮助程序员将精力集中于所要完成的任务上, 先编写单元测试案例然后编写代码或许能够导致更快地完成工作, 比直接编写代码更快。用极限编程的术语来说就是:

测试程序 + 编码 比直接编码更快。

先编写测试程序同样能够防止边界条件破坏程序, 使程序的代码更加健壮。

如果系统无法正常工作, 而代码却通过了所有的测试程序, 那么问题多半不是出在代码中。“代码已经通过了所有测试程序”就是一个很好的理由。

### 2.2.1 自动测试

单元测试是什么样子的呢? 有太多的开发人员常常只希望他们的代码在获得符合要求的输入时能够产生预期的输出, 他们只是用眼睛检查程序的输出。这种方法存在两个危险。首先, 程序的输入不可能总是符合要求。大家都知道应该检查程序输入数据的边界, 但是当程序员竭尽全力来使程序能够工作时, 很难顾及考虑这些事情。如果在编写程序代码之前首先编写测试程序, 就可以以测试工程师的角度来问自己, “什么情况会造成程序的破坏呢?” 编写测试程序能够证明所写的函数不会被破坏掉, 然后程序员再以开发者的身份来完成这些函数。首先编写测试程序能够使程序员写出更好的代码。

第二个危险是用眼睛观察来检查程序的输出, 这是很乏味的而且容易出错。这样的事情计算机也能做, 并且不会出错。最好用布尔表达式的集合来表示测试问题, 让测试程序来报告编码的任何错误。

例如, 假设想要构造一个 **Date** 类, 这个类有以下的特性:

- 可以用一个字符串 (YYYYMMDD)、三个整数 (Y, M, D) 或者什么也不用 (获得当前日期) 来初始化日期值。
- **Date** 对象能够生成年、月、日的值或 “YYYYMMDD” 形式的字符串。
- 所有相关量能够进行有效的比较、能够计算两个日期的差 (在年、月、日中)。
- 日期的比较能够跨越任意个世纪 (例如, 1600 和 2200)。

这个类能够用三个整数分别表示年、月、日。(确保表示年的整数至少是 16 位 (bit) 的, 以满足上面所说的最后一个特性。) **Date** 类的接口可能如下所示:

```
//: C02:Date1.h
// A first pass at Date.h.
#ifndef DATE1_H
#define DATE1_H
#include <string>
```

① 参考 Kent Beck 所著《Extreme Programming Explained: Embrace Change》, Addison Wesley 1999。轻量级方法, 例如 XP 已经使 the Agile Alliance 得到了加强 (参见 <http://www.agilealliance.org/home>)。



```

class Date {
public:
    // A struct to hold elapsed time:
    struct Duration {
        int years;
        int months;
        int days;
        Duration(int y, int m, int d)
            : years(y), months(m), days(d) {}
    };
    Date();
    Date(int year, int month, int day);
    Date(const std::string&);
    int getYear() const;
    int getMonth() const;
    int getDay() const;
    std::string toString() const;
    friend bool operator<(const Date&, const Date&);
    friend bool operator>(const Date&, const Date&);
    friend bool operator<=(const Date&, const Date&);
    friend bool operator>=(const Date&, const Date&);
    friend bool operator==(const Date&, const Date&);
    friend bool operator!=(const Date&, const Date&);
    friend Duration duration(const Date&, const Date&);
};
#endif // DATE1_H ///:~

```

在实现这个类之前，读者可以先编写测试程序，使读者牢固地掌握需求。读者可能提出如下代码：

```

//: C02:SimpleDateTest.cpp
//{L} Date
#include <iostream>
#include "Date.h" // From Appendix B
using namespace std;

// Test machinery
int nPass = 0, nFail = 0;
void test(bool t) { if(t) nPass++; else nFail++; }

int main() {
    Date mybday(1951, 10, 1);
    test(mybday.getYear() == 1951);
    test(mybday.getMonth() == 10);
    test(mybday.getDay() == 1);
    cout << "Passed: " << nPass << ", Failed: "
         << nFail << endl;
}
/* Expected output:
Passed: 3, Failed: 0
*/ ///:~

```

在这个普通的测试案例中，函数**test()**维护着两个全局变量**nPass**和**nFail**。惟一需要程序员用眼睛检查的是最终的得分结果。如果测试失败，更复杂的**test()**函数能够显示适当的消息。在这一章的后面描述的测试框架不但包括这样一个测试函数，而且还包括其他一些东西。

现在，可以逐步实现**Date**类，使其通过测试，然后可以继续进行反复测试直到满足所有需求。由于是首先编写测试程序，程序员会更多地注意考虑其他边边角角的情况，而这些情况可能破坏即将实现的程序，会使程序员在第一时间就更加注意编写出正确的代码。这样的练习

可能会使读者写出下面的用于测试**Date**类的一种描述：

```
//: C02:SimpleDateTest2.cpp
//{L} Date
#include <iostream>
#include "Date.h"
using namespace std;

// Test machinery
int nPass = 0, nFail = 0;
void test(bool t) { if(t) ++nPass; else ++nFail; }

int main() {
    Date mybday(1951, 10, 1);
    Date today;
    Date myeveday("19510930");

    // Test the operators
    test(mybday < today);
    test(mybday <= today);
    test(mybday != today);
    test(mybday == mybday);
    test(mybday >= mybday);
    test(mybday <= mybday);
    test(myeveday < mybday);
    test(mybday > myeveday);
    test(mybday >= myeveday);
    test(mybday != myeveday);

    // Test the functions
    test(mybday.getYear() == 1951);
    test(mybday.getMonth() == 10);
    test(mybday.getDay() == 1);
    test(myeveday.getYear() == 1951);
    test(myeveday.getMonth() == 9);
    test(myeveday.getDay() == 30);
    test(mybday.toString() == "19511001");
    test(myeveday.toString() == "19510930");

    // Test duration
    Date d2(2003, 7, 4);
    Date::Duration dur = duration(mybday, d2);
    test(dur.years == 51);
    test(dur.months == 9);
    test(dur.days == 3);

    // Report results:
    cout << "Passed: " << nPass << ", Failed: "
         << nFail << endl;
} ///:~
```

这个测试案例可以开发得更加完整。例如，在这里还没有测试程序的可持续性。至此作者所要表达的意思应该已经很清楚了。**Date**类的完整实现在附录中的**Date.h**和**Date.cpp**文件中<sup>①</sup>。

### 2.2.2 TestSuite框架

读者可以从万维网（World Wide Web）下载某些C++自动单元测试工具，例如

① 本书所写的**Date**类是能够国际化的，也就是说它支持宽字符集（wide character set）。我们将在下一章的结尾介绍宽字符集。

**CppUnit**。<sup>①</sup>在这里,本节的目的是不仅仅为了介绍一种易于使用的测试结构,而且要使读者容易理解它,甚至在必要的时候修改它。因此,怀着“只要能用,做最简单的”的信念,<sup>②</sup>作者开发了测试套件框架 (TestSuite Framework),从其名字的命名就可以看出**TestSuite**中包含两个主要的类:**Test**和**Suite**。

**Test**类是一个抽象基类,可以从这个类派生用户自己的测试对象。**Test**类保存着测试时成功和失败的次数,测试失败时能够显示相关测试条件等信息。只需重写成员函数**run()**就行了,在这个函数中应该定义一些布尔型的测试条件,并且依次调用**test\_()**宏来测试它们。

为了使用这个框架来定义测试**Date**类的案例,可以继承**Test**类,下面的程序就是这个测试案例:

```
//: C02:DateTest.h
#ifndef DATETEST_H
#define DATETEST_H
#include "Date.h"
#include "../TestSuite/Test.h"

class DateTest : public TestSuite::Test {
    Date mybday;
    Date today;
    Date myeveday;
public:
    DateTest(): mybday(1951, 10, 1), myeveday("19510930") {}
    void run() {
        testOps();
        testFunctions();
        testDuration();
    }
    void testOps() {
        test_(mybday < today);
        test_(mybday <= today);
        test_(mybday != today);
        test_(mybday == mybday);
        test_(mybday >= mybday);
        test_(mybday <= mybday);
        test_(myeveday < mybday);
        test_(mybday > myeveday);
        test_(mybday >= myeveday);
        test_(mybday != myeveday);
    }
    void testFunctions() {
        test_(mybday.getYear() == 1951);
        test_(mybday.getMonth() == 10);
        test_(mybday.getDay() == 1);
        test_(myeveday.getYear() == 1951);
        test_(myeveday.getMonth() == 9);
        test_(myeveday.getDay() == 30);
        test_(mybday.toString() == "19511001");
        test_(myeveday.toString() == "19510930");
    }
    void testDuration() {
        Date d2(2003, 7, 4);
        Date::Duration dur = duration(mybday, d2);
        test_(dur.years == 51);
        test_(dur.months == 9);
    }
};
```

① 参考<http://sourceforge.net/projects/cppunit>可以得到更多信息。

② 这是极限编程的主要原则之一。



```

        test_(dur.days == 3);
    }
};
#endif // DATETEST_H ///:~

```

运行这个测试案例很简单，只需实例化一个**DateTest**对象并调用它的成员函数**run()**就可以了。

```

//: C02:DateTest.cpp
// Automated testing (with a framework).
//{L} Date ../TestSuite/Test
#include <iostream>
#include "DateTest.h"
using namespace std;

int main() {
    DateTest test;
    test.run();
    return test.report();
}
/* Output:
Test "DateTest":
    Passed: 21,      Failed: 0
*/ ///:~

```

**Test::report()**函数显示前面的输出，并且把测试失败的次数作为返回值，这个值也适合作为**main()**函数的返回值。

**Test**类使用运行时类型识别（**RTTI**）<sup>①</sup>来取得测试类的类名（例如，**DateTest**），并将这个类名用于测试结果报告。默认情况下**Test**类将测试结果送到标准输出，如果想要把测试结果写到文件中可以使用**setStream()**成员函数。在本章的后面，读者将会看到**Test**类的实现。

**test\_()**宏能够将失败的布尔条件摘录成文本形式，并且使这段文本包含**test\_()**宏所在文件的文件名和**test\_()**宏所在行的行号。<sup>②</sup>为了观察测试失败时会发生什么情况，可以在代码中故意引入错误，例如：可以颠倒上一个例子代码中**DateTest::testOps()**函数在第一次调用**test\_()**时所用的测试条件。程序的输出准确地显示了在哪里、哪个测试出现了错误。

```

DateTest failure: (mybday > today) , DateTest.h (line 31)
Test "DateTest":
    Passed: 20      Failed: 1

```

除了**test\_()**之外，框架中还包括函数**succeed\_()**和**fail\_()**，这两个函数用于无法使用布尔测试的情况。当测试类的时候可能抛出异常的，这时候应该使用这两个函数。在测试的时候创建一个会触发异常的输入集。如果异常没有发生，就表明程序中出现了错误，这时候应该调用**fail\_()**，就可以清楚地显示一段消息并且修改测试失败次数的值。如果期望的异常发生了，就应该调用**succeed\_()**修改测试成功次数的值。

为了举例说明这两种情况，假设现在已经修改了**Date**类两个非默认构造函数的异常规格说明。如果输入参数不能表示一个合法的日期，这两个构造函数会抛出**DateError**异常（嵌套在**Date**类内的一个类型，派生自**std::logic\_error**）：

① “运行时类型识别”将在第9章中讨论。使用**typeid**类的**name()**成员函数。如果读者使用Microsoft Visual C++，必须指定一个编译器选项 **/GR**。如果没有指定这个参数，在运行时将会出现非法访问错误。

② 使用字符串化运算（stringizing，通过**#**预处理运算符）以及预定义的宏**\_FILE\_**和**\_LINE\_**。参考本章随后部分的代码。

```
Date(const string& s) throw(DateError);
Date(int year, int month, int day) throw(DateError);
```

现在, 成员函数**DateTest::run()**可以调用下面的函数来测试异常处理了:

```
void testExceptions() {
    try {
        Date d(0,0,0); // Invalid
        fail_("Invalid date undetected in Date int ctor");
    } catch(Date::DateError&) {
        succeed_();
    }
    try {
        Date d(""); // Invalid
        fail_("Invalid date undetected in Date string ctor");
    } catch(Date::DateError&) {
        succeed_();
    }
}
```

在这两种情况下, 如果函数中不抛出异常, 就表明程序出错了。注意, 由于这种测试不计算布尔表达式的值, 所以必须用手工方式向**fail\_()**中传递一个消息作为参数。

### 2.2.3 测试套件

实际的软件项目通常包含很多类, 需要一种组织测试用例的方式, 使程序设计人员能够通过按一个按钮来测试整个项目。<sup>①</sup> **Suite**类可以将测试案例集中到一个函数单元中。程序设计人员可以使用**addTest()**成员函数添加一个**Test**对象到**Suite**中, 也可以使用**addSuite()**将现有的一个测试套件添加到**Suite**中。为了演示**Suite**的使用, 将第3章中用到**Test**类的程序集中到一个单独的测试套件中。注意, 这些文件在第3章的文件子目录中。

```
//: C03:StringSuite.cpp
//{L} ../TestSuite/Test ../TestSuite/Suite
//{L} TrimTest
// Illustrates a test suite for code from Chapter 3
#include <iostream>
#include "../TestSuite/Suite.h"
#include "StringStorage.h"
#include "Sieve.h"
#include "Find.h"
#include "Rparse.h"
#include "TrimTest.h"
#include "CompStr.h"
using namespace std;
using namespace TestSuite;

int main() {
    Suite suite("String Tests");
    suite.addTest(new StringStorageTest);
    suite.addTest(new SieveTest);
    suite.addTest(new FindTest);
    suite.addTest(new RparseTest);
    suite.addTest(new TrimTest);
    suite.addTest(new CompStrTest);
    suite.run();
    long nFail = suite.report();
    suite.free();
}
```

① 也可以使用批处理文件和shell脚本文件。**Suite**类是一种基于C++的组织相关测试用例的方法。

```

        return nFail;
    }
    /* Output:
    s1 = 62345
    s2 = 12345
    Suite "String Tests"
    =====
    Test "StringStorageTest":
        Passed: 2   Failed: 0
    Test "SieveTest":
        Passed: 50  Failed: 0
    Test "FindTest":
        Passed: 9   Failed: 0
    Test "RparseTest":
        Passed: 8   Failed: 0
    Test "TrimTest":
        Passed: 11  Failed: 0
    Test "CompStrTest":
        Passed: 8   Failed: 0
    */ ///:~

```

上述的5个测试案例完全包含在头文件中。因为**TrimTest**包含一个静态数据，而静态数据必须定义在实现文件中，所以**TrimTest**不但需要一个头文件，而且还需要实现文件。程序输出的头两行是**StringStorage**测试的结果。程序员必须向测试套件的构造函数传递一个参数，这个参数就是测试套件的名字。成员函数**Suite::run()**调用它所包含的每一个测试案例的**Test::run()**函数。**Suite::report()**函数所做的工作与此差不多，也可以将每个测试案例的测试报告输出到不同的流中，而不使用那个属于测试套件的报告。如果用**addSuite()**添加的测试案例已经被指定了流指针，那么这个测试案例将使用这个流。否则，测试案例使用**Suite**对象指定的输出流。（就像**Test**一样，测试套件的构造函数有1个可选的第2个参数，这个参数的默认值为**std::cout**。）**Suite**的析构函数并不能自动删除它包含的指向**Test**对象的指针，因为这些**Test**对象并不需要保留在堆上；而这项工作由**Suite::free()**来完成。

#### 2.2.4 测试框架的源代码

在代码解压包中，测试框架的源代码包含在一个叫做**TestSuite**的文件子目录中，可以从[www.MindView.net](http://www.MindView.net)网站上得到这个代码的解压包。为了使用这个测试框架，读者必须在头文件的查找路径中包含**TestSuite**子目录，在库文件的查找路径中包含**TestSuite**子目录，这样才能链接相关的目标文件。下面是**Test.h**头文件：

```

//: TestSuite:Test.h
#ifndef TEST_H
#define TEST_H
#include <string>
#include <iostream>
#include <cassert>
using std::string;
using std::ostream;
using std::cout;

// fail_() has an underscore to prevent collision with
// ios::fail(). For consistency, test_() and succeed_()
// also have underscores.

#define test_(cond) \
    do_test(cond, #cond, __FILE__, __LINE__)
#define fail_(str) \
    do_fail(str, __FILE__, __LINE__)

```



```

namespace TestSuite {

class Test {
    ostream* osptr;
    long nPass;
    long nFail;
    // Disallowed:
    Test(const Test&);
    Test& operator=(const Test&);
protected:
    void do_test(bool cond, const string& lbl,
        const char* fname, long lineno);
    void do_fail(const string& lbl,
        const char* fname, long lineno);
public:
    Test(ostream* osptr = &cout) {
        this->osptr = osptr;
        nPass = nFail = 0;
    }
    virtual ~Test() {}
    virtual void run() = 0;
    long getNumPassed() const { return nPass; }
    long getNumFailed() const { return nFail; }
    const ostream* getStream() const { return osptr; }
    void setStream(ostream* osptr) { this->osptr = osptr; }
    void succeed_() { ++nPass; }
    long report() const;
    virtual void reset() { nPass = nFail = 0; }
};

} // namespace TestSuite
#endif // TEST_H ///:~

```

**Test**类有3个虚函数：

- 虚析构造函数
- **reset()**函数
- 纯虚函数**run()**

我们在第1卷中曾说过，通过基类指针释放在堆上分配的派生类对象是错误的，除非基类有1个虚析构造函数。任何想成为基类的类（如果类中出现了至少1个其他虚函数，就说明这个类想成为基类）应该有1个虚的析构造函数。**Test::reset()**的默认实现只是将成功和失败计数器的值重置为零。读者可以重写这个函数，让它重置派生测试对象中数据的状态；确保在重写函数中明确调用**Test::reset()**，使其重置计数器的状态。因为需要在派生类中重写**Test::run()**函数，所以它是一个纯虚成员函数。

在预处理的时候，**test\_()**和**fail\_()**宏能够取得其所在文件的文件名和其所在行的行号。刚开始的时候并没有在这两个宏的名字后面加下划线，但是**fail()**宏与**ios::fail()**产生冲突，造成了编译器错误。

下面是**Test**类其余函数的实现：

```

//: TestSuite:Test.cpp {0}
#include "Test.h"
#include <iostream>
#include <typeinfo>
using namespace std;
using namespace TestSuite;

void Test::do_test(bool cond, const std::string& lbl,

```

```

    const char* fname, long lineno) {
        if(!cond)
            do_fail(lbl, fname, lineno);
        else
            succeed_();
    }

    void Test::do_fail(const std::string& lbl,
        const char* fname, long lineno) {
        ++nFail;
        if(osptr) {
            *osptr << typeid(*this).name()
                << "failure: (" << lbl << ") , " << fname
                << " (line " << lineno << ")" << endl;
        }
    }

    long Test::report() const {
        if(osptr) {
            *osptr << "Test \"" << typeid(*this).name()
                << "\":\n\tPassed: " << nPass
                << "\tFailed: " << nFail
                << endl;
        }
        return nFail;
    } ///:~

```

**Test**类不但保存着成功测试的次数和失败测试的次数，而且保存着**Test::report()**显示测试结果所需的流。**test\_()**和**fail\_()**宏在预处理的时候取得当前文件的文件名和当前行的行号信息，并把文件名传递给**do\_test()**，把行号传递给**do\_fail()**，这两个函数则显示一个消息并修改相关的计数器。我们认为测试对象没有理由使用拷贝和赋值操作，所以通过将这两个函数的原型声明为私有并且忽略他们各自的函数体来禁止这两种操作。

下面是**Suite**类的头文件：

```

//: TestSuite:Suite.h
#ifndef SUITE_H
#define SUITE_H
#include <vector>
#include <stdexcept>
#include "../TestSuite/Test.h"
using std::vector;
using std::logic_error;

namespace TestSuite {

class TestSuiteError : public logic_error {
public:
    TestSuiteError(const string& s = "")
        : logic_error(s) {}
};

class Suite {
    string name;
    ostream* osptr;
    vector<Test*> tests;
    void reset();
    // Disallowed ops:
    Suite(const Suite&);
    Suite& operator=(const Suite&);
public:

```





```

Suite(const string& name, ostream* osptr = &cout)
: name(name) { this->osptr = osptr; }
string getName() const { return name; }
long getNumPassed() const;
long getNumFailed() const;
const ostream* getStream() const { return osptr; }
void setStream(ostream* osptr) { this->osptr = osptr; }
void addTest(Test* t) throw(TestSuiteError);
void addSuite(const Suite&);
void run(); // Calls Test::run() repeatedly
long report() const;
void free(); // Deletes tests
};

} // namespace TestSuite
#endif // SUITE_H ///:~

```

**Suite**类在**vector**中保存指向**Test**对象的指针。请注意**addTest()**成员函数上的异常规格说明。当读者向测试套件中添加一个测试案例的时候，**Suite::addTest()**检查传递到这个函数的指针是否为空；如果为空指针，则抛出**TestSuiteError**异常。由于在这种情况下不可能把一个空指针传递给测试套件，所以**addSuite()**用断言检查测试套件所包含的每一个测试案例，就像其他函数遍历测试案例的**vector**一样（请参考下面的实现）。像**Test**类一样，**Suite**类禁止拷贝构造函数和赋值操作。

```

//: TestSuite:Suite.cpp {0}
#include "Suite.h"
#include <iostream>
#include <cassert>
#include <cstdint>
using namespace std;
using namespace TestSuite;

void Suite::addTest(Test* t) throw(TestSuiteError) {
    // Verify test is valid and has a stream:
    if(t == 0)
        throw TestSuiteError("Null test in Suite::addTest");
    else if(osptr && !t->getStream())
        t->setStream(osptr);
    tests.push_back(t);
    t->reset();
}

void Suite::addSuite(const Suite& s) {
    for(size_t i = 0; i < s.tests.size(); ++i) {
        assert(tests[i]);
        addTest(s.tests[i]);
    }
}

void Suite::free() {
    for(size_t i = 0; i < tests.size(); ++i) {
        delete tests[i];
        tests[i] = 0;
    }
}

void Suite::run() {
    reset();
    for(size_t i = 0; i < tests.size(); ++i) {
        assert(tests[i]);
    }
}

```



```

        tests[i]->run();
    }
}

long Suite::report() const {
    if(osptr) {
        long totFail = 0;
        *osptr << "Suite \"" << name
                << "\"\n=====";
        size_t i;
        for(i = 0; i < name.size(); ++i)
            *osptr << '=';
        *osptr << "\n" << endl;
        for(i = 0; i < tests.size(); ++i) {
            assert(tests[i]);
            totFail += tests[i]->report();
        }
        *osptr << "=====";
        for(i = 0; i < name.size(); ++i)
            *osptr << '=';
        *osptr << "\n" << endl;
        return totFail;
    }
    else
        return getNumFailed();
}

long Suite::getNumPassed() const {
    long totPass = 0;
    for(size_t i = 0; i < tests.size(); ++i) {
        assert(tests[i]);
        totPass += tests[i]->getNumPassed();
    }
    return totPass;
}

long Suite::getNumFailed() const {
    long totFail = 0;
    for(size_t i = 0; i < tests.size(); ++i) {
        assert(tests[i]);
        totFail += tests[i]->getNumFailed();
    }
    return totFail;
}

void Suite::reset() {
    for(size_t i = 0; i < tests.size(); ++i) {
        assert(tests[i]);
        tests[i]->reset();
    }
}
} ///:~

```

在本教材的剩余部分中将使用**TestSuite**框架。

## 2.3 调试技术

最好的调试习惯是使用本章开始的时候所述的断言；使用断言可以在程序代码真正出现问题之前，帮助程序设计人员找到其中的逻辑错误。这部分内容介绍的一些技巧和技术可以在程序调试过程中给予一定的帮助。

### 2.3.1 用于代码跟踪的宏

某些情况下，在程序执行过程中将执行到的每一条语句行代码打印到**cout**或一个跟踪文件中是很有用处的。下面是一个能够完成这种功能的预处理宏：

```
#define TRACE(ARG) cout << #ARG << endl; ARG
```

现在可以用这个宏来处理跟踪语句代码了，然而这可能会导致一些问题。例如，如果采用下面的语句：

```
for(int i = 0; i < 100; i++)
    cout << i << endl;
```

将这两行程序都放在**TRACE()**宏中，就会得到如下代码：

```
TRACE(for(int i = 0; i < 100; i++))
TRACE( cout << i << endl;)
```

预处理之后，代码会变成这样：

```
cout << "for(int i = 0; i < 100; i++)" << endl;
for(int i = 0; i < 100; i++)
    cout << "cout << i << endl;" << endl;
cout << i << endl;
```

这并不是我们想要的。因此，使用这种技术时必须格外细心。

下面是**TRACE()**宏的一个变种：

```
#define D(a) cout << #a "=[" << a << "]" << endl;
```

如果要想显示一个表达式，只需用它作为参数调用**D()**。程序执行时会显示这个表达式，接着显示它的值（假设这个表达式的结果类型重载了运算符**<<**）。例如，可以这样写**D(a + b)**。并可以在任何时间使用这个宏来检查中间结果的值。

这两个宏能够完成调试器所能实现的两个最基本功能：跟踪代码的执行过程并显示表达式的值。好的调试器是个杰出且高效的工具，但是在某些时候，可能找不到可以使用的调试器，或者找到的调试器不好用。但是不管在任何情况下，读者都能使用上述两种技术。

### 2.3.2 跟踪文件

**免责声明：**这部分和下面部分内容所包含的代码还尚未被C++标准正式接受。特别是，这里使用宏重定义了**cout**和**new**，如果不仔细的话可能会造成奇怪的结果。这里提供的例子可以在作者使用的所有编译器中正常工作，并提供有用的信息。这是本教材唯一一处偏离编码实践兼容性标准的地方。是否使用在于读者自己！注意，为了使这个例子能够工作，必须使用**using**声明，这样可以去掉**cout**前面的名字空间前缀，例如，在这段代码中不能使用**std::cout**。

下面的代码简单地创建了一个跟踪文件，并把所有本来应被送到**cout**的输出送到了这个跟踪文件。现在必须做的所有事情就是**#define TRACEON**并且包含相关的头文件（当然，仅仅将两行关键代码正确地写到文件中是相当的容易）。

```
//: C03:Trace.h
// Creating a trace file.
#ifndef TRACE_H
#define TRACE_H
#include <fstream>

#ifdef TRACEON
std::ofstream TRACEFILE__("TRACE.OUT");
#define cout TRACEFILE__
#endif
```

```
#endif // TRACE_H ///:~
```

下面这段代码是对上述头文件的简单测试：

```
//: C03:Tracetst.cpp {-bor}
#include <iostream>
#include <fstream>
#include "../require.h"
using namespace std;

#define TRACEON
#include "Trace.h"

int main() {
    ifstream f("Tracetst.cpp");
    assure(f, "Tracetst.cpp");
    cout << f.rdbuf(); // Dumps file contents to file
} ///:~
```

因为`cout`已经被**Trace.h**中的宏修改成了其他东西，所以程序中所有的`cout`语句现在都把信息送到了跟踪文件。当读者所使用的操作系统不能简便的进行输出重定向时，这种方式能够方便地将输出保存到文件中。

### 2.3.3 发现内存泄漏

第1卷中讲解过下列直观的调试技术：

1) 为了对数组边界进行检查，可以使用第1卷**C16:Array3.cpp**中实现的**Array**模板来定义所有数组。当准备发行代码的时候，可以关闭边界检查以提高性能。（尽管这种方法对于指针数组不管用。）

2) 检查基类中的非虚析构函数。

**跟踪new/delete和malloc/free语句**

通常的内存分配问题包括：对不是在动态存储区（free store）上分配的内存误使用**delete**，多次重复释放在动态存储区上分配的一个内存，最常见的情况是忘记删除一个指针。这一节讨论了一种能够帮助跟踪这类问题的系统。

上一小节所述免责声明的附加条款：因为这种方法重载了运算符**new**，所以下述技术在某些平台上可能无法使用，而且只能用于不直接调用**operator new()**函数的程序。在这本教材中，我们一直非常小心，希望只介绍完全符合C++标准的代码，但是这是一个特例，主要基于如下原因：

1) 尽管这种技术是不合标准的，但是它能用于很多编译器。<sup>①</sup>

2) 我们希望利用这种方法来阐述某些有用的思想。

为了使用这种内存检查系统，在这里只需包含头文件**MemCheck.h**，并链接**MemCheck.obj**到应用程序中，这个系统能够截获所有对**new**和**delete**的调用，并且通过在程序中调用宏**MEM\_ON()**（在本章的后面解释）来初始化内存跟踪。所有有关内存分配和释放的踪迹都被打印在标准输出上（通过**stdout**）。当使用这种系统的时候，**new**运算符所在文件的文件名和**new**运算符所在行的行号被保存了下来。这是用**operator new**的定位语法（placement syntax）来完成的。<sup>②</sup>虽然在典型的情况下，只有当需要将一个对象放到内存中

① 本书主要的技术审阅者，Dinkumware公司的Pete Becker，提醒读者使用宏来替换C++关键字是不符合规定的。他对这种技术的评价是：“这是一种旁门左道（dirty trick）。有时候必须利用旁门左道来找出代码不能正确运行的原因，所以可以把它放到我们的工具箱中，但是不要把它留在发行版本中。”这是对程序员的告诫。

② 感谢C++标准委员会的成员Reg Charney提出这种诀窍。

的指定位置时才使用定位语法。这种内存检查方法也可以创建带有多个参数的**operator new( )**来达到目的。下面的例子就是用有多个参数的**operator new( )**来实现的，当**new**被调用的时候，用**\_FILE\_**和**\_LINE\_**宏来获得其所在的文件名和行号并存储：

```
//: C02:MemCheck.h
#ifndef MEMCHECK_H
#define MEMCHECK_H
#include <stddef> // For size_t

// Usurp the new operator (both scalar and array versions)
void* operator new(std::size_t, const char*, long);
void* operator new[](std::size_t, const char*, long);
#define new new (__FILE__, __LINE__)

extern bool traceFlag;
#define TRACE_ON() traceFlag = true
#define TRACE_OFF() traceFlag = false

extern bool activeFlag;
#define MEM_ON() activeFlag = true
#define MEM_OFF() activeFlag = false

#endif // MEMCHECK_H ///:~
```

重要的是，当读者想跟踪动态存储区的活动时，可以在任何源文件中包含这个文件，但是它必须是所有被包含文件的最后一个（在其他**#include**之后）。标准库中大部分头文件定义的是模板类，并且大多数编译器使用模板编译的包含模型（inclusion model）（这句话的意思是说，所有源代码都包含在头文件中），**MemCheck.h**中替换**new**的宏将会篡改库中源代码所使用的所有**new**运算符的实例（并且可能造成编译错误）。另外，读者大概只想跟踪存在于自己编写的代码中的内存错误，而不会理会库中的代码是不是有错。

下面的文件包含内存跟踪的实现，所有的输出都是通过C的标准输入/输出来完成的，而没有使用C++的输入输出流。这样做没有什么差别，虽然对动态存储区使用输入输出流也不会受到干扰，但是当尝试使用输入输出流时，有些编译器会报错。而所有编译器都能接受**<cstdio>**版本的输入输出。

```
//: C02:MemCheck.cpp {0}
#include <cstdio>
#include <cstdlib>
#include <cassert>
#include <stddef>
using namespace std;
#undef new

// Global flags set by macros in MemCheck.h
bool traceFlag = true;
bool activeFlag = false;

namespace {

// Memory map entry type
struct Info {
    void* ptr;
    const char* file;
    long line;
};

// Memory map data
```



```

const size_t MAXPTRS = 10000u;
Info memMap[MAXPTRS];
size_t nptrs = 0;

// Searches the map for an address
int findPtr(void* p) {
    for(size_t i = 0; i < nptrs; ++i)
        if(memMap[i].ptr == p)
            return i;
    return -1;
}

void delPtr(void* p) {
    int pos = findPtr(p);
    assert(pos >= 0);
    // Remove pointer from map
    for(size_t i = pos; i < nptrs-1; ++i)
        memMap[i] = memMap[i+1];
    --nptrs;
}

// Dummy type for static destructor
struct Sentinel {
    ~Sentinel() {
        if(nptrs > 0) {
            printf("Leaked memory at:\n");
            for(size_t i = 0; i < nptrs; ++i)
                printf("\t%p (file: %s, line %ld)\n",
                    memMap[i].ptr, memMap[i].file, memMap[i].line);
        }
        else
            printf("No user memory leaks!\n");
    }
};

// Static dummy object
Sentinel s;

} // End anonymous namespace

// Overload scalar new
void*
operator new(size_t siz, const char* file, long line) {
    void* p = malloc(siz);
    if(activeFlag) {
        if(nptrs == MAXPTRS) {
            printf("memory map too small (increase MAXPTRS)\n");
            exit(1);
        }
        memMap[nptrs].ptr = p;
        memMap[nptrs].file = file;
        memMap[nptrs].line = line;
        ++nptrs;
    }
    if(traceFlag) {
        printf("Allocated %u bytes at address %p ", siz, p);
        printf("(file: %s, line: %ld)\n", file, line);
    }
    return p;
}

// Overload array new
void*

```

```

operator new[](size_t siz, const char* file, long line) {
    return operator new(siz, file, line);
}

// Override scalar delete
void operator delete(void* p) {
    if(findPtr(p) >= 0) {
        free(p);
        assert(nptrs > 0);
        delPtr(p);
        if(traceFlag)
            printf("Deleted memory at address %p\n", p);
    }
    else if(!p && activeFlag)
        printf("Attempt to delete unknown pointer: %p\n", p);
}

// Override array delete
void operator delete[](void* p) {
    operator delete(p);
} ///:~

```

布尔型标志**traceFlag**和**activeFlag**是全局变量，可以在代码中用宏**TRACE\_ON()**、**TRACE\_OFF()**、**MEM\_ON()**和**MEM\_OFF()**来修改它们。一般来说，可以用**MEM\_ON()**和**MEM\_OFF()**这对宏将**main()**函数中的所有代码包围起来，这样内存的分配和释放就会一直被跟踪。内存跟踪显示了函数**operator new()**和**operator delete()**的活动。这种跟踪在默认情况下是打开的，可以用**TRACE\_OFF()**来关闭它。任何情况下，最终结果都会打印出来（参考本章后面的测试运行）。

**MemCheck**工具在**Info**结构类型的数组中保存全部内存地址、文件名和行号：内存地址是使用**operator new()**分配内存时得到的，文件名是**new**运算符所在文件的文件名，而行号是**new**运算符所在行的行号。为了避免与放入全局名字空间中的其他名字冲突，应该把尽可能多的内容放在匿名名字空间中。当程序停止的时候，单独存在的**Sentinel**类调用一个静态对象的析构函数。这个析构函数检查**memMap**，看看是否有等待删除的指针（表明程序中存在内存泄漏）。

在程序中，**operator new()**使用**malloc()**来获取内存，然后把指针和相关的文件信息保存到**memMap**中。**operator delete()**函数做相反的工作，它调用**free()**释放内存并将**nptrs**的值减1，但是它首先会检查传送过来的指针参数是否在映射表（map）中。如果这个指针不在映射表中，就说明程序员正在试图释放的不是在动态存储区上分配的内存，或者已经释放了这段内存，并把这段内存的地址从映射表中删除了。**activeFlag**变量在这里非常重要，因为不想对系统关闭过程中所做的内存释放活动进行处置。通过在程序代码的最后调用**MEM\_OFF()**可以将**activeFlag**设为**false**，这样，随后的**delete**调用将会被忽略。（在实际的程序中，这样做是不好的，但是，在这里这样做的目的是发现内存泄漏，而不是调试库。）简单地说，现在做的所有工作就是排列**new**和**delete**，将它们进行匹配。

下面是一个使用**MemCheck**工具进行测试的简单例子：

```

//: C02:MemTest.cpp
//{L} MemCheck
// Test of MemCheck system.
#include <iostream>
#include <vector>
#include <cstring>
#include "MemCheck.h" // Must appear last!

```

```

using namespace std;

class Foo {
    char* s;
public:
    Foo(const char*s ) {
        this->s = new char[strlen(s) + 1];
        strcpy(this->s, s);
    }
    ~Foo() { delete [] s; }
};

int main() {
    MEM_ON();
    cout << "hello" << endl;
    int* p = new int;
    delete p;
    int* q = new int[3];
    delete [] q;
    int* r;
    delete r;
    vector<int> v;
    v.push_back(1);
    Foo s("goodbye");
    MEM_OFF();
} ///:~

```

这个例子证实了，可以在如下场合中使用**MemCheck**：代码中使用了流，代码中使用了标准容器（standard containers），以及代码中某个类的构造函数分配了内存。指针**p**和**q**的内存分配和释放没有问题，但是指针**r**不是指向在堆上分配的内存的指针，所以程序的输出显示了一个错误，报告程序试图删除一个未知的指针。

```

hello
Allocated 4 bytes at address 0xa010778 (file: memtest.cpp,
line: 25)
Deleted memory at address 0xa010778
Allocated 12 bytes at address 0xa010778 (file: memtest.cpp,
line: 27)
Deleted memory at address 0xa010778
Attempt to delete unknown pointer: 0x1
Allocated 8 bytes at address 0xa0108c0 (file: memtest.cpp,
line: 14)
Deleted memory at address 0xa0108c0
No user memory leaks!

```

因为调用了**MEM\_OFF()**，所以后面**vector**和**ostream**对**operator delete()**的调用过程并没有进行。读者仍然可能会见到容器重新分配内存时调用**delete**所产生的输出结果。

如果在程序的开始就调用**TRACE\_OFF()**，那么输出结果将是：

```

hello
Attempt to delete unknown pointer: 0x1
No user memory leaks!

```

## 2.4 小结

令软件工程师头痛的大多数问题都可以通过仔细考虑正在进行的工作来避免。即使没有按常规在代码中使用**assert()**宏，在编写循环和函数的时候，程序设计人员还是会在心里使用断言。如果使用**assert()**，将会很快发现程序中存在的逻辑错误，并且最终能够写出可读性



更好的程序。记住，断言只能用于不变量条件检查，而不能将其用于运行时错误处理。

没有什么能够比彻底测试完代码更能够给软件开发者的的心灵带来如此的安宁了。如果在过去的时候（程序中的错误使）人心生烦恼，那么就使用自动测试框架——像教材中介绍的这种——把常规的测试集成到自己的日常工作中吧。软件开发（和他们的用户）将会为其所做的工作而感到高兴。

## 2.5 练习

2-1 使用**TestSuite**框架编写一个测试程序来测试标准**vector**类，彻底地测试整型**vector**类的下列成员函数：**push\_back()**（在**vector**的末端添加一个元素）、**front()**（返回**vector**中的第一个元素）、**back()**（返回**vector**中的最后一个元素）、**pop\_back()**（删除最后一个元素，不返回它）、**at()**（返回指定索引位置中的元素）和**size()**（返回元素的个数）。验证：如果给出的索引产生越界情况，**vector::at()**会抛出**std::out\_of\_range**异常。

2-2 假设有人要求开发一个名为**Rational**的类，这个类支持有理数（分数）。在**Rational**对象中的分数始终保存最低项（默认值为0），并且分母为0的情况是一个错误。下面是**Rational**类接口的例子：

```
//: C02:Rational.h {-xo}
#ifndef RATIONAL_H
#define RATIONAL_H
#include <iosfwd>

class Rational {
public:
    Rational(int numerator = 0, int denominator = 1);
    Rational operator-() const;
    friend Rational operator+(const Rational&,
                             const Rational&);
    friend Rational operator-(const Rational&,
                             const Rational&);
    friend Rational operator*(const Rational&,
                             const Rational&);
    friend Rational operator/(const Rational&,
                             const Rational&);

    friend std::ostream&
    operator<<(std::ostream&, const Rational&);
    friend std::istream&
    operator>>(std::istream&, Rational&);
    Rational& operator+=(const Rational&);
    Rational& operator-=(const Rational&);
    Rational& operator*=(const Rational&);
    Rational& operator/=(const Rational&);
    friend bool operator<(const Rational&,
                          const Rational&);
    friend bool operator>(const Rational&,
                          const Rational&);
    friend bool operator<=(const Rational&,
                           const Rational&);
    friend bool operator>=(const Rational&,
                           const Rational&);
    friend bool operator==(const Rational&,
                           const Rational&);
    friend bool operator!=(const Rational&,
                           const Rational&);
```



```
};
#endif // RATIONAL_H ///:~
```

为这个类编写一个完整的规格说明，包括前置条件、后置条件和异常说明。

- 2-3 使用**TestSuite**框架编写一个测试案例，为上一个练习写出的所有规格说明做彻底地测试，包括测试异常。
- 2-4 实现**Rational**类，使其通过上一个练习时写出的所有测试案例。仅对不变量使用断言。
- 2-5 下面的**BuggedSearch.cpp**文件包含一个二分查找函数，这个函数在区间[**beg**, **end**)中查找整型数**what**。算法中有些错误。使用本章介绍的跟踪技术调试这个查找函数。

```
//: C02:BuggedSearch.cpp {-xo}
//{L} ../TestSuite/Test
#include <cstdlib>
#include <ctime>
#include <cassert>
#include <fstream>
#include "../TestSuite/Test.h"
using namespace std;

// This function is only one with bugs
int* binarySearch(int* beg, int* end, int what) {
    while(end - beg != 1) {
        if(*beg == what) return beg;
        int mid = (end - beg) / 2;
        if(what <= beg[mid]) end = beg + mid;
        else beg = beg + mid;
    }
    return 0;
}

class BinarySearchTest : public TestSuite::Test {
    enum { SZ = 10 };
    int* data;
    int max; // Track largest number
    int current; // Current non-contained number
                // Used in notContained()
    // Find the next number not contained in the array
    int notContained() {
        while(data[current] + 1 == data[current + 1])
            ++current;
        if(current >= SZ) return max + 1;
        int retValue = data[current++] + 1;
        return retValue;
    }
    void setData() {
        data = new int[SZ];
        assert(!max);
        // Input values with increments of one. Leave
        // out some values on both odd and even indexes.
        for(int i = 0; i < SZ;
            rand() % 2 == 0 ? max += 1 : max += 2)
            data[i++] = max;
    }
    void testInBound() {
        // Test locations both odd and even
        // not contained and contained
        for(int i = SZ; --i >= 0;)
            test_(binarySearch(data, data + SZ, data[i]));
        for(int i = notContained(); i < max;
            i = notContained())
            test_(!binarySearch(data, data + SZ, i));
    }
}
```



```
void testOutBounds() {
    // Test lower values
    for(int i = data[0]; --i > data[0] - 100;)
        test_(!binarySearch(data, data + SZ, i));
    // Test higher values
    for(int i = data[SZ - 1];
        ++i < data[SZ - 1] + 100;)
        test_(!binarySearch(data, data + SZ, i));
}
public:
    BinarySearchTest() { max = current = 0; }
    void run() {
        setData();
        testInBound();
        testOutBounds();
        delete [] data;
    }
};

int main() {
    srand(time(0));
    BinarySearchTest t;
    t.run();
    return t.report();
} ///:~
```



## 第二部分 标准C++库

标准C++除了包含所有的标准C库外（其中有为支持类型安全而进行的少许增加与更改），还加进了自己特有的库。这些库比起标准C中的库，功能更加强大。可以说，它们的影响力几乎就等同于由C到C++的转变。

本教材的这一部分将对标准C++库的重点部分进行深入的介绍。

有关整个C++库的最全面，也是最令人费解的参考读物就是C++标准本身。Bjarne Stroustrup编写的《The C++ Programming Language》<sup>①</sup>（第3版，Addison Wesley, 2000）对C++语言和库来说仍然是值得信赖的参考读物。最著名的专门论述C++库的参考读物是Nicolai Josuttis的《The C++ Standard Library: A Tutorial and Reference》（Addison Wesley, 1999）。本部分中各章的编写目的是：给读者提供丰富的描述说明与范例，使读者在解决与标准库的使用相关的任何问题时都有一个好的起点。但是，这里没有涉及某些不常用的技术和主题。如果在这些章里没有找到所需的内容，请参考上述两本书。编写这本教材的目的不是替代上述两本书，而是提供一些必要的补充。希望读者学习完接下来的内容后能够更轻松地理解那两本书。

读者会发现，这些章里并不包含标准C++库中的所有函数和类的详细文档，本教材把这些描述工作留给了别人，特别是P. J. Plauger的《Dinkumware C/C++ Library Reference》，在<http://www.dinkumware.com>可以得到这些文档。它是用超文本标记语言（Hypertext Markup Language, HTML）格式写成的，是标准库文档联机资源中的精品。读者可守在电脑旁，当需要查找某些内容时，使用一下网络浏览器即可查到。可以联机查阅，也可以购买这些文档放在本机上，以便随时查阅。它包括C和C++库的全部参考页（reference page）。（所以，它能够很好地帮助读者解决有关标准C/C++编程的问题。）电子文档是十分有效的，因为它不但随时可用，而且还能进行电子查找。

以上这些资料可满足程序员在奋力编程时的参考需要（如果本书对某些内容讲述不清，也可以参考上述这些资料）。附录A也列举了一些其他的参考资料。

这部分的首章介绍了标准C++ **string**类。它是一个功能非常强大的工具，可以简化在文本处理中可能遇到的大部分“琐事”。很可能在C语言中需要使用多行代码才能完成的字符串处理操作，用**string**类中的一个成员函数调用就可以完成。

第4章介绍的内容是**iostreams**库，它的内容包含与文件、字符串对象（string target）和系统控制台（system console）输入输出操作相关的类。

虽然第5章“深入理解模板”不是完全针对库的一章，但它对后面两章的内容介绍做了必要的准备工作。第6章将研究标准C++库提供的通用算法。由于这些算法都是用模板实现的，因此可以将它们应用于任意对象序列（sequence）。第7章介绍了标准容器及它们关联的迭代器（associated iterator）。首先介绍算法的原因是，只使用数组和**vector**容器（从第1卷开始就一直在使用）就可对其进行全面的研究。很自然地，本教材会在与容器相关的部分使用标准算法，因此在研究容器前熟悉算法是很有好处的。

---

① 本书已由机械工业出版社引进出版。——编辑注

## 深入理解字符串

在C语言中，对字符型数组进行字符串处理是最费时的工作之一。字符型数组要求程序员了解静态引用串（static quoted string）与在堆和堆栈中生成的数组之间的差别，实际上有时用类型“**char\***”就能达到要求，而有时则必须拷贝整个数组。

尤其是，由于字符串操作的普遍性，字符型数组可能造成许多混淆与错误。尽管如此，多年来创建字符串类仍是初级C++程序员通常的练习题。标准C++库中的**string**类一劳永逸地解决了字符型数组的处理问题，它监控内存存在空间分配和拷贝构造时的情况，程序设计人员根本就不用为此劳神。

本章<sup>①</sup>研究标准C++中的**string**类，先简要介绍C++字符串的构成要素，然后阐释C++版本的字符串类与传统C语言字符型数组有哪些不同。读者将会了解使用**string**对象时的各种操作方法，还会看到C++ **string**类在处理不同字符集和字符串数据转换时的神来之笔。

文本处理是编程语言最古老的应用之一，因此C++ **string**类吸取了大量曾经被C及其他编程语言长时间使用的编程思想和术语。当开始介绍C++ **string**类时，应该再次明确这个事实。不论采用哪一种编程方法，有3个操作是我们希望**string**类能够做到的：

- 创建或修改**string**中存放的字符序列。
- 检测**string**中元素的存在性。
- 能够在多种描述**string**字符的方案之间进行转换。

读者将会看到C++ **string**对象是怎样完成这些工作的。

### 3.1 字符串的内部是什么

在C语言中，字符串基本上就是字符型数组，并且总是以二进制零（通常被称为空结束符（null terminator））作为其最末元素。C++ **string**与它们在C语言中的前身截然不同。首先，也是最重要的不同点，C++ **string**隐藏了它所包含的字符序列的物理表示。程序设计人员不必关心数组的维数或空结束符方面的问题。**string**也包含关于其数据容量及存储地址的“内存处理”信息。具体地说，C++ **string**对象知道自己在内存中的开始位置、包含的内容、包含的字符长度（length in characters）以及在必需重新调整内部数据缓冲区的大小之前自己可以增长到的最大字符长度。C++字符串极大地减少了C语言编程中3种最常见且最具破坏性的错误：超越数组边界，通过未被初始化或被赋以错误值的指针来访问数组元素，以及在释放了某一数组原先所分配的存储单元后仍保留了“悬挂”指针。

C++标准没有定义字符串类内存布局（memory layout）的确切实现。采用这种体系结构是为了获得足够的灵活性，从而允许不同的编译器厂商能够提供不同的实现，并且向用户保证提供可预测的行为。特别是，C++标准没有定义在何种确切的情况下应该为字符串对象分配存储单元来保存数据。字符串分配规则明确规定：允许但不要求引用计数实现（reference-counted implementation），但无论其实现是否采用引用计数（reference counting），其语义都必须一致。

① 本章的某些材料来源于Nancy Nicolaisen。

这种表示稍有不同，在C语言中，每个**char**型数组都占据各自的物理存储区。在C++中，独立的几个**string**对象可以占据也可以不占据各自特定的物理存储区，但是，如果采用引用计数避免了保存同一数据的拷贝副本，那么各个独立的对象（在处理上）必须看起来并表现得就像独占地拥有各自的存储区一样。例如：

```
//: C03:StringStorage.h
#ifndef STRINGSTORAGE_H
#define STRINGSTORAGE_H
#include <iostream>
#include <string>
#include "../TestSuite/Test.h"
using std::cout;
using std::endl;
using std::string;

class StringStorageTest : public TestSuite::Test {
public:
    void run() {
        string s1("12345");
        // This may copy the first to the second or
        // use reference counting to simulate a copy:
        string s2 = s1;
        test_(s1 == s2);
        // Either way, this statement must ONLY modify s1:
        s1[0] = '6';
        cout << "s1 = " << s1 << endl; // 62345
        cout << "s2 = " << s2 << endl; // 12345
        test_(s1 != s2);
    }
};
#endif // STRINGSTORAGE_H ///:~

//: C03:StringStorage.cpp
//{L} ../TestSuite/Test
#include "StringStorage.h"

int main() {
    StringStorageTest t;
    t.run();
    return t.report();
} ///:~
```

只有当字符串被修改的时候才创建各自的拷贝，这种实现方式称为写时复制（copy-on-write）策略。当字符串只是作为值参数（value parameter）或在其他只读情形下使用，这种方法能够节省时间和空间。

不论一个库的实现是不是采用引用计数，它对**string**类的使用者来说都应该是透明的。遗憾的是，情况并不总是这样。在多线程<sup>①</sup>程序中，几乎不可能安全地使用引用计数来实现。

## 3.2 创建并初始化C++字符串

创建和初始化字符串对象是一件简单的事情并且相当灵活。在下面的**Smallstring.cpp**例子中，第1个**string**对象**imBlank**虽然被声明了，但并不包含初始值。C语言中的**char**型数组在初始化前都包含随机的无意义的位模式（bit pattern），而与此不同，**imBlank**确实包

① 很难在保证线程安全的前提下实现引用计数（参阅Herb Sutter的《More Exceptional C++》第104~114页）。详见第11章关于多线程编程的部分。

含了有意义的信息。这个**string**对象被初始化成包含“没有字符 (no character)”，通过类的成员函数能够正确地报告其长度为零并且没有数据元素。

第2个串是**heyMom**，它被文字参数“Where are my socks?”初始化，这种形式的初始化使用一个引用字符数组 (quoted character array) 作为**string**构造函数的参数。相比之下，对象**standardReply**只使用一个赋值操作来完成初始化。这一组中的最后一个字符串是**use ThisOneAgain**，它的初始化采用的是一个现有的C++**string**对象来完成。换句话说，这个例子阐述了可以对新创建的**string**对象做以下几件事：

- 创建空**string**对象，且并不立即用字符数据对其初始化。
- 将一个文字的引用字符数组作为参数传递给构造函数，以此来对一个**string**对象进行初始化。
- 用等号(=)来初始化一个**string**对象。
- 用一个**string**对象初始化另一个**string**对象。

```
//: C03:SmallString.cpp
#include <string>
using namespace std;

int main() {
    string imBlank;
    string heyMom("Where are my socks?");
    string standardReply = "Beamed into deep "
        "space on wide angle dispersion?";
    string useThisOneAgain(standardReply);
} ///:~
```

这些都是**string**对象初始化最简单的形式，但若对此做少许改动，便可更灵活地进行初始化，并对其进行更好地控制。可以这样做：

- 使用C语言的**char**型数组或C++ **string**类两者任一个的一部分。
- 用**operator+**来将不同的初始化数据源结合在一起。
- 用**string**对象的成员函数**substr()**来创建一个子串。

下面的程序解释了这些特征：

```
//: C03:SmallString2.cpp
#include <string>
#include <iostream>
using namespace std;

int main() {
    string s1("What is the sound of one clam napping?");
    string s2("Anything worth doing is worth overdoing.");
    string s3("I saw Elvis in a UFO");
    // Copy the first 8 chars:
    string s4(s1, 0, 8);
    cout << s4 << endl;
    // Copy 6 chars from the middle of the source:
    string s5(s2, 15, 6);
    cout << s5 << endl;
    // Copy from middle to end:
    string s6(s3, 6, 15);
    cout << s6 << endl;
    // Copy many different things:
    string quoteMe = s4 + "that" +
        // substr() copies 10 chars at element 20
        s1.substr(20, 10) + s5 +
```



```
// substr() copies up to either 100 char
// or eos starting at element 5
"with" + s3.substr(5, 100) +
// OK to copy a single char this way
s1.substr(37, 1);
cout << quoteMe << endl;
} ///:~
```

**string**类对象的成员函数**substr()**将开始位置作为其第1个参数，而将待选字符的个数作为其第2个参数。两个参数都有默认值。如果使用空的参数列表来调用**substr()**，那么将会构造出整个**string**对象的一个拷贝，所以这是复制**string**对象的一种简便方法。

下面是程序的输出：

```
What is
doing
Elvis in a UFO
What is that one clam doing with Elvis in a UFO?
```

注意上例的最后一行。C++允许不同的**string**类对象初始化技术在单个语句中的混合使用，这是一种灵活方便的特征。还有，最后一个初始化操作从源**string**对象中复制的仅仅是一个字符。

另一个稍微精巧些的初始化方法利用了**string**类的迭代器**string::begin()**和**string::end()**。这种技术将**string**看做容器对象（迄今为止读者所见到的容器主要是**vector**——在第7章将会看到更多的容器），它用迭代器来指示字符序列的开始与结尾。借助这种方法，就可以给**string**类的构造函数传递两个迭代器，构造函数从一个迭代器开始直到另一个迭代器结束，将它们之间的数据拷贝到新的**string**对象中：

```
//: C03:StringIterators.cpp
#include <string>
#include <iostream>
#include <cassert>
using namespace std;

int main() {
    string source("xxx");
    string s(source.begin(), source.end());
    assert(s == source);
} ///:~
```

迭代器并不局限于**begin()**和**end()**；可以对一个对象使用的迭代器的运算包括增1、减1以及加上整数偏移量，这些运算允许程序员从源**string**对象中提取字符的子集。

不可以使用单个的字符、ASCII码或其他整数值来初始化C++字符串。但是，可用单个字符的多个拷贝来初始化字符串：

```
//: C03:UhOh.cpp
#include <string>
#include <cassert>
using namespace std;

int main() {
    // Error: no single char inits
    ///! string nothingDoing1('a');
    // Error: no integer inits
    ///! string nothingDoing2(0x37);
    // The following is legal:
    string okay(5, 'a');
    assert(okay == string("aaaaa"));
} ///:~
```





第1个参数表示放入字符串中的第2个参数的拷贝的个数。第2个参数只能是单个字符的 **char** 型数据，而不能是 **char** 型数组。

### 3.3 对字符串进行操作

有用C语言编程经验的人，都习惯用函数族对 **char** 型数组进行写入、查找、修改和复制等操作。对于 **char** 型数组的处理，标准C语言库函数中有两个方面不太尽如人意。首先，这些函数分为两族 (family)，组织得十分松散：无格式 (plain) 族，以及那些在随后的操作中需要提供计算字符个数的函数族。C语言提供的用于处理 **char** 型字符串的那些库函数的函数名列表不但冗长，而且充满了模糊不清、晦涩难懂的名字，其中大部分的名字叫人读不出来，这些都让虔诚的用户很吃惊。虽然这些函数的参数类型及个数颇为一致，但想要用好这些函数，程序员必须对函数命名和参数传递的细节等慎之又慎。

标准C语言的 **char** 型数组工具中存在着其固有的第2个误区，那就是它们都显式地依赖一种假设：字符串包括一个空结束符。若由于疏忽或是其他差错，这个空结束符被忽略或重写，这个小小的差错就会使C语言的 **char** 型数组处理函数几乎不可避免地操作其已分配空间之外的内存，有时会带来灾难性的后果。

C++提供的 **string** 类对象，在使用的便利性和安全性上都有很大的提高。为了实际的字符串处理操作，在 **string** 类中，不同名的成员函数的数量几乎跟C语言库中的函数一样多，但是由于有重载，使 **string** 类的功能更加强大。这些特征再加上C++命名机制理性化以及明智地使用了默认参数，使 **string** 类比起C语言库的 **char** 型数组函数更便于使用。

#### 3.3.1 追加、插入和连接字符串

C++字符串有几个颇具价值而且最便于使用的特色，其中之一就是：无需程序员干预，它们可根据需要自行扩充规模。这不仅使得字符串处理代码更加可靠，同时也几乎完全消除了令人生厌的“内存处理”琐事——跟踪字符串的存储边界。比方说，创建一个字符串对象并且将其初始化为一个由50个‘X’组成的字符串，然后再存进50个“Zowie”，这个字符串对象自己会自动重新分配足够的存储空间来适应数据的增长。如果代码处理的字符串改变了长度，但程序员并不知道改变的幅度，也许只有这时读者才能最真切地感受到C++字符串的优越性。此外，当字符串增长时，字符串成员函数 **append()** 和 **insert()** 很明显地重新分配了存储空间：

```
//: C03:StrSize.cpp
#include <string>
#include <iostream>
using namespace std;

int main() {
    string bigNews("I saw Elvis in a UFO. ");
    cout << bigNews << endl;
    // How much data have we actually got?
    cout << "Size = " << bigNews.size() << endl;
    // How much can we store without reallocating?
    cout << "Capacity = " << bigNews.capacity() << endl;
    // Insert this string in bigNews immediately
    // before bigNews[1]:
    bigNews.insert(1, " thought I");
    cout << bigNews << endl;
    cout << "Size = " << bigNews.size() << endl;
    cout << "Capacity = " << bigNews.capacity() << endl;
    // Make sure that there will be this much space
    bigNews.reserve(500);
```



```
// Add this to the end of the string:
bigNews.append("I've been working too hard.");
cout << bigNews << endl;
cout << "Size = " << bigNews.size() << endl;
cout << "Capacity = " << bigNews.capacity() << endl;
} ///:~
```

下面是来自特定编译器的输出：

```
I saw Elvis in a UFO.
Size = 22
Capacity = 31
I thought I saw Elvis in a UFO.
Size = 32
Capacity = 47
I thought I saw Elvis in a UFO. I've been
working too hard.
Size = 59
Capacity = 511
```

这个例子证实了，即使可以安全地避免分配及管理**string**对象所占用的存储空间的工作，C++**string**类也提供了几个工具以便监视和管理它们的存储规模。注意到改变分配给字符串的存储空间的规模是多么轻松了吧。**size()**函数返回当前在字符串存储的字符数，它跟**length()**成员函数的作用是一样的。**capacity()**函数返回当前分配的存储空间的规模，也即在没有要求更多存储空间时，字符串所能容纳的最大字符数。**reserve()**函数提供一种优化机制，它按照程序员的意图，预留一定数量的存储空间，以便将来使用；**capacity()**返回的值不小于最近一次调用**reserve()**所使用的值。如果要生成的新字符串的规模比当前的字符串大或者说是需要截短原字符串，**resize()**函数就会在字符串的末尾追加空格。（**resize()**的一个重载可以指定一个不同的填充字符。）

**string**类的成员函数为数据分配存储空间的确切方式取决于C++类库的实现。在使用C++类库的某种实现来测试上述例子时，读者会发现，当系统进行存储空间再分配遇到偶数字（word）（即，全整数（full-integer））的边界时，会隐含增加一个字节。为什么会这样呢？**string**类的设计者曾做过不懈的努力让**char**型数组和C++字符串对象可以混合使用，为此，在这种特定的实现中，**StrSize.cpp**报告的存储容量数字，意味着预留出一个字节以便很容易地容纳空结束符（用**char**型数组表示一个字符串时，该字符串的最后一个表示串结束的字符）的插入。

### 3.3.2 替换字符串中的字符

**insert()**函数使程序员放心地向字符串中插入字符，而不必担心会使存储空间越界，或者会改写插入点之后紧跟的字符。存储空间增大了，原有的字符会很“礼貌地”改变其存储位置，以便安置新元素。但有时这并不是程序员所希望的。如果希望字符串的大小保持不变，就应该使用**replace()**函数来改写字符。**replace()**有很多的重载版本，最简单的版本用了3个参数：一个参数用于指示从字符串的什么位置开始改写；第二个参数用于指示从原字符串中剔除多少个字符；另外一个替换字符串（它所包含的字符数可以与被剔除的字符数不同）。举例如下：

```
///C03:StringReplace.cpp
// Simple find-and-replace in strings.
#include <cassert>
#include <string>
using namespace std;

int main() {
```

```

string s("A piece of text");
string tag("$tag$");
s.insert(8, tag + ' ');
assert(s == "A piece $tag$ of text");
int start = s.find(tag);
assert(start == 8);
assert(tag.size() == 5);
s.replace(start, tag.size(), "hello there");
assert(s == "A piece hello there of text");
} ///:~

```

**tag**串首先插入到**s**串中（注意：在函数调用中的第1个参数值指示的插入点之前进行插入，并且在**tag**串后添加一个额外的空字符），接着进行查找和替换。

在调用**replace()**前程序员应检查是否会找到什么。前面的例子用一个**char\***来进行替换操作，**replace()**还有一个重载版本，用一个**string**来进行替换操作。下面的例子更完整地演示了**replace()**函数：

```

//: C03:Replace.cpp
#include <cassert>
#include <cstdint> // For size_t
#include <string>
using namespace std;

void replaceChars(string& modifyMe,
    const string& findMe, const string& newChars) {
    // Look in modifyMe for the "find string"
    // starting at position 0:
    size_t i = modifyMe.find(findMe, 0);
    // Did we find the string to replace?
    if(i != string::npos)
        // Replace the find string with newChars:
        modifyMe.replace(i, findMe.size(), newChars);
}

int main() {
    string bigNews = "I thought I saw Elvis in a UFO. "
        "I have been working too hard.";
    string replacement("wig");
    string findMe("UFO");
    // Find "UFO" in bigNews and overwrite it:
    replaceChars(bigNews, findMe, replacement);
    assert(bigNews == "I thought I saw Elvis in a "
        "wig. I have been working too hard.");
} ///:~

```

如果**replace**找不到要查找的字符串，它返回 **string::npos**。数据成员**npos**是**string**类的一个静态常量成员，它表示一个不存在的字符位置。<sup>①</sup>

当有新字符复制到现存的一串序列的元素中间时，**replace()**并不增加**string**的存储空间规模，这一点与**insert()**不同。但是，**replace()**必要时也会增加存储空间，例如当所做的“替换”会使原字符串扩充到超越当前分配的存储边界时。举例如下：

```

//: C03:ReplaceAndGrow.cpp
#include <cassert>
#include <string>

```

① 它是“无位置”（no position）的缩写，并且是字符串分配算符**size\_type**（默认是**std::size\_t**）所能表示的最大值。

```
using namespace std;

int main() {
    string bigNews("I have been working the grave.");
    string replacement("yard shift.");
    // The first argument says "replace chars
    // beyond the end of the existing string":
    bigNews.replace(bigNews.size() - 1,
        replacement.size(), replacement);
    assert(bigNews == "I have been working the "
        "graveyard shift.");
} ///:~
```

对**replace()**的调用使“替换”超出了原有序列的边界，这与追加操作是等价的。注意，此例中**replace()**扩展了相应的串序列的规模。

读者可能会不辞劳苦地研读本章，试图找到相对简单的题目，如用一个字符替换字符串中各处出现的另一不同字符。一旦找到前面这些关于替换的材料，读者就会认为找到了答案，然后就开始学习貌似很复杂的材料，如替换字符组和计数等。难道**string**类就没有一种方法用一个字符替换字符串中各处出现的另一个字符吗？

借助如下的**find()**和**replace()**成员函数，可很容易地实现上述函数：

```
//: C03:ReplaceAll.h
#ifndef REPLACEALL_H
#define REPLACEALL_H
#include <string>

std::string& replaceAll(std::string& context,
    const std::string& from, const std::string& to);
#endif // REPLACEALL_H ///:~

//: C03:ReplaceAll.cpp {0}
#include <cstdint>
#include "ReplaceAll.h"
using namespace std;

string& replaceAll(string& context, const string& from,
    const string& to) {
    size_t lookHere = 0;
    size_t foundHere;
    while((foundHere = context.find(from, lookHere))
        != string::npos) {
        context.replace(foundHere, from.size(), to);
        lookHere = foundHere + to.size();
    }
    return context;
} ///:~
```

此处使用的**find()**版本将开始查找的位置作为第2参数，如果找不到则返回**string::npos**。将变量**lookHere**表示的位置传送到替换串，这是很重要的，以防字符串**from**是字符串**to**的子串。下面的程序测试了**replaceAll**函数：

```
//: C03:ReplaceAllTest.cpp
//{L} ReplaceAll
#include <cassert>
#include <iostream>
#include <string>
#include "ReplaceAll.h"
using namespace std;
```

```
int main() {
    string text = "a man, a plan, a canal, Panama";
    replaceAll(text, "an", "XXX");
    assert(text == "a mXXX, a plXXX, a cXXXa1, PXXXama");
} ///:~
```

大家知道，**string**类自身并不能解决所有可能出现的问题。许多解决方案都是由标准库<sup>①</sup>中的算法完成的，因为**string**类几乎可与**STL**序列等价（借助于前面所说的迭代器）。所有通用算法的工作对象都是容器中某个“范围”内的元素。通常这个范围指的是“从容器前端到末尾”。**string**对象看上去就像是字符的容器：可用**string::begin()**得到容器范围的前端，用**string::end()**得到其末尾。下面的例子显示了如何使用**replace()**算法将所有单个的字符‘X’替换为‘Y’：

```
///: C03:StringCharReplace.cpp
#include <algorithm>
#include <cassert>
#include <string>
using namespace std;

int main() {
    string s("aaaXaaaXXaaXXXaXXXaXaa");
    replace(s.begin(), s.end(), 'X', 'Y');
    assert(s == "aaaYaaaYYaaYYYaYYYaXaa");
} ///:~
```

注意，这里调用的**replace()**并不是**string**的成员函数。另外，**replace()**算法将字符串中出现的某个字符全部用另一个字符替换掉，这一点与**string::replace()**函数不同，因为后者只进行一次替换。

**replace()**算法的工作对象只是单一的对象（本例中是**char**对象），它不会替换引用**char**型数组或**string**对象。由于**string**很像一个**STL**序列，很多其他算法对它也适用，这些算法可以解决**string**类的成员函数没能直接解决的问题。

### 3.3.3 使用非成员重载运算符连接

对于一个学习C++**string**处理的C程序员来说，等待他的最令人欣喜的发现之一就是，借助**operator+**和**operator+=**可以如此轻而易举地实现**string**的合并与追加。这些运算符使合并串的操作在语法上类似于数值型数据的加法运算：

```
///: C03:AddStrings.cpp
#include <string>
#include <cassert>
using namespace std;

int main() {
    string s1("This ");
    string s2("That ");
    string s3("The other ");
    // operator+ concatenates strings
    s1 = s1 + s2;
    assert(s1 == "This That ");
    // Another way to concatenates strings
    s1 += s3;
    assert(s1 == "This That The other ");
}
```

① 将在第6章详述。



```
// You can index the string on the right
s1 += s3 + s3[4] + "ooh lala";
assert(s1 == "This That The other The other oooh lala");
} ///:~
```

使用**operator+**和**operator+=**运算符是合并**string**数据的一种既灵活又方便的方法。在语句的右边，程序员几乎可以采用任意一种样式对由单字符或多字符构成的分组进行赋值。

### 3.4 字符串的查找

**string**成员函数中的**find**族是用来在给定字符串中定位某个或某组字符的。下面是**find**族成员及其一般用法：

字符串查找成员函数	函数功能及实现
<b>find()</b>	在一个字符串中查找一个指定的单个字符或字符组。如果找到，就返回首次匹配的起始位置；如果没有查找到匹配的内容，则返回 <b>npos</b>
<b>find_first_of()</b>	在一个目标串中进行查找，返回值是第1个与指定字符组中任何字符匹配的字符位置。如果没有查找到匹配的内容，则返回 <b>npos</b>
<b>find_last_of()</b>	在一个目标串中进行查找，返回最后一个与指定字符组中任何字符匹配的字符位置。如果没有查找到匹配的内容，则返回 <b>npos</b>
<b>find_first_not_of()</b>	在一个目标串中进行查找，返回第一个与指定字符组中任何字符都不匹配的元素的位置。如果找不到那样的元素则返回 <b>npos</b>
<b>find_last_not_of()</b>	在一个目标串中进行查找，返回下标值最大的与指定字符组中任何字符都不匹配的元素的位置。若找不到那样的元素则返回 <b>npos</b>
<b>rfind()</b>	对一个串从尾至头查找一个指定的单个字符或字符组。如果找到，就返回首次匹配的起始位置。如果没有查找到匹配的内容，则返回 <b>npos</b>

**find()**的最简单应用就是在**string**对象中查找一个或多个字符。这个重载的**find()**函数使用一个参数用来指示要查找的字符（子串），还有另一可选的参数用来表示从字符串的何处开始查找子串。（默认的开始查找位置是0。）把**find**放在循环体内，可以很容易地从头至尾遍历一个字符串，重复查找字符串中所有可能出现的与指定字符或字符组匹配的子串。

下面的程序使用Eratosthenes筛选法查找小于50的素数。这种方法从数字2开始，标记所有2（3，5，…）的倍数为非素数，对其他后选素数重复该过程。**SieveTest**的构造函数对**sieveChars**进行初始化，设置其字符序列（array）的初始大小，并且用‘P’来填充每个成员。

```
//: C03:Sieve.h
#ifndef SIEVE_H
#define SIEVE_H
#include <cmath>
#include <cstdint>
#include <string>
#include "../TestSuite/Test.h"
using std::size_t;
using std::sqrt;
using std::string;

class SieveTest : public TestSuite::Test {
    string sieveChars;
```

```

public:
    // Create a 50 char string and set each
    // element to 'P' for Prime:
    SieveTest() : sieveChars(50, 'P') {}
    void run() {
        findPrimes();
        testPrimes();
    }
    bool isPrime(int p) {
        if(p == 0 || p == 1) return false;
        int root = int(sqrt(double(p)));
        for(int i = 2; i <= root; ++i)
            if(p % i == 0) return false;
        return true;
    }
    void findPrimes() {
        // By definition neither 0 nor 1 is prime.
        // Change these elements to "N" for Not Prime:
        sieveChars.replace(0, 2, "NN");
        // Walk through the array:
        size_t sieveSize = sieveChars.size();
        int root = int(sqrt(double(sieveSize)));
        for(int i = 2; i <= root; ++i)
            // Find all the multiples:
            for(size_t factor = 2; factor * i < sieveSize;
                ++factor)
                sieveChars[factor * i] = 'N';
    }
    void testPrimes() {
        size_t i = sieveChars.find('P');
        while(i != string::npos) {
            test_(isPrime(i++));
            i = sieveChars.find('P', i);
        }
        i = sieveChars.find_first_not_of('P');
        while(i != string::npos) {
            test_(!isPrime(i++));
            i = sieveChars.find_first_not_of('P', i);
        }
    }
};
#endif // SIEVE_H ///:~

//: C03:Sieve.cpp
//{L} ../TestSuite/Test
#include "Sieve.h"

int main() {
    SieveTest t;
    t.run();
    return t.report();
} ///:~

```

**find()**函数在**string**内部进行搜索，检测多次出现的一个字符或字符组，**find\_first\_not\_of()**查找其他的字符或子串。

**string**类中没有改变字符串大小写的函数，但借助于标准C语言的库函数**toupper()**和**tolower()**（这两个函数一次只改变一个字符的大小写），可很容易地创建这类函数。下面的例子演示了忽略了大小写的查找：

```

//: C03:Find.h
#ifndef FIND_H
#define FIND_H
#include <cctype>

```

```

#include <cstddef>
#include <string>
#include "../TestSuite/Test.h"
using std::size_t;
using std::string;
using std::tolower;
using std::toupper;

// Make an uppercase copy of s
inline string upperCase(const string& s) {
    string upper(s);
    for(size_t i = 0; i < s.length(); ++i)
        upper[i] = toupper(upper[i]);
    return upper;
}

// Make a lowercase copy of s
inline string lowerCase(const string& s) {
    string lower(s);
    for(size_t i = 0; i < s.length(); ++i)
        lower[i] = tolower(lower[i]);
    return lower;
}

class FindTest : public TestSuite::Test {
    string chooseOne;
public:
    FindTest() : chooseOne("Eenie, Meenie, Miney, Mo") {}
    void testUpper() {
        string upper = upperCase(chooseOne);
        const string LOWER = "abcdefghijklmnopqrstuvwxyz";
        test_(upper.find_first_of(LOWER) == string::npos);
    }
    void testLower() {
        string lower = lowerCase(chooseOne);
        const string UPPER = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
        test_(lower.find_first_of(UPPER) == string::npos);
    }
    void testSearch() {
        // Case sensitive search
        size_t i = chooseOne.find("een");
        test_(i == 8);
        // Search lowercase:
        string test = lowerCase(chooseOne);
        i = test.find("een");
        test_(i == 0);
        i = test.find("een", ++i);
        test_(i == 8);
        i = test.find("een", ++i);
        test_(i == string::npos);
        // Search uppercase:
        test = upperCase(chooseOne);
        i = test.find("EEN");
        test_(i == 0);
        i = test.find("EEN", ++i);
        test_(i == 8);
        i = test.find("EEN", ++i);
        test_(i == string::npos);
    }
    void run() {
        testUpper();
        testLower();
        testSearch();
    }
}

```





```

    }
};
#endif // FIND_H ///:~

//: C03:Find.cpp
//{L} ../TestSuite/Test
#include "Find.h"
#include "../TestSuite/Test.h"

int main() {
    FindTest t;
    t.run();
    return t.report();
} ///:~

```

**upperCase()**和**lowerCase()**两个函数的流程形式相同：它们先复制参数**string**对象，接着改变其大小写。程序**Find.cpp**并不是解决大小写敏感问题的最佳方案，所以在讲到**string**的比较时将会再次讨论它。

### 3.4.1 反向查找

如果需要在**string**对象中从后往前进行查找（用“后进／先出”的顺序查找数据），可以使用字符串成员函数**rfind()**：

```

//: C03:Rparse.h
#ifndef RPARSE_H
#define RPARSE_H
#include <cstddef>
#include <string>
#include <vector>
#include "../TestSuite/Test.h"
using std::size_t;
using std::string;
using std::vector;

class RparseTest : public TestSuite::Test {
    // To store the words:
    vector<string> strings;
public:
    void parseForData() {
        // The ';' characters will be delimiters
        string s("now.;sense;make;to;going;is;This");
        // The last element of the string:
        int last = s.size();
        // The beginning of the current word:
        size_t current = s.rfind(';');
        // Walk backward through the string:
        while(current != string::npos) {
            // Push each word into the vector.
            // Current is incremented before copying
            // to avoid copying the delimiter:
            ++current;
            strings.push_back(s.substr(current, last - current));
            // Back over the delimiter we just found,
            // and set last to the end of the next word:
            current -= 2;
            last = current + 1;
            // Find the next delimiter:
            current = s.rfind(';', current);
        }
        // Pick up the first word -- it's not
        // preceded by a delimiter:
    }
};

```



```

        strings.push_back(s.substr(0, last));
    }
    void testData() {
        // Test them in the new order:
        test_(strings[0] == "This");
        test_(strings[1] == "is");
        test_(strings[2] == "going");
        test_(strings[3] == "to");
        test_(strings[4] == "make");
        test_(strings[5] == "sense");
        test_(strings[6] == "now.");
        string sentence;
        for(size_t i = 0; i < strings.size() - 1; i++)
            sentence += strings[i] += " ";
        // Manually put last word in to avoid an extra space:
        sentence += strings[strings.size() - 1];
        test_(sentence == "This is going to make sense now.");
    }
    void run() {
        parseForData();
        testData();
    }
};
#endif // RPARSE_H ///:~

//: C03:Rparse.cpp
//{L} ../TestSuite/Test
#include "Rparse.h"

int main() {
    RparseTest t;
    t.run();
    return t.report();
} ///:~

```

字符串成员函数**rfind()**从后往前遍历字符串，查找并且报告与其匹配字符（组）所在的序列排列（array）下标，若不成功则报告**string::npos**。

### 3.4.2 查找一组字符第1次或最后一次出现的位置

使用**find\_first\_of()**和**find\_last\_of()**成员函数可以很方便地实现一些小的功能，比如从字符串的头尾两端删除空白字符。注意，它并不触动原字符串，而是返回一个新字符串：

```

//: C03:Trim.h
// General tool to strip spaces from both ends.
#ifndef TRIM_H
#define TRIM_H
#include <string>
#include <cstdint>

inline std::string trim(const std::string& s) {
    if(s.length() == 0)
        return s;
    std::size_t beg = s.find_first_not_of(" \\a\b\f\n\r\t\v");
    std::size_t end = s.find_last_not_of(" \\a\b\f\n\r\t\v");
    if(beg == std::string::npos) // No non-spaces
        return "";
    return std::string(s, beg, end - beg + 1);
}
#endif // TRIM_H ///:~

```

第1次条件判断是为了检查**string**是否为空；如果为空，则直接返回原字符串的1个拷贝，

不再进行其他判断。注意，一旦找到结束点，函数就会使用开始点的位置和计算出来的子串长度作为参数调用**string**类的构造函数，用来创建1个基于原字符串的新的**string**对象。

对这样一个通用工具进行的测试需要十分彻底：

```
//: C03:TrimTest.h
#ifndef TRIMTEST_H
#define TRIMTEST_H
#include "Trim.h"
#include "../TestSuite/Test.h"

class TrimTest : public TestSuite::Test {
    enum {NTESTS = 11};
    static std::string s[NTESTS];
public:
    void testTrim() {
        test_(trim(s[0]) == "abcdefghijklmno");
        test_(trim(s[1]) == "abcdefghijklmno");
        test_(trim(s[2]) == "abcdefghijklmno");
        test_(trim(s[3]) == "a");
        test_(trim(s[4]) == "ab");
        test_(trim(s[5]) == "abc");
        test_(trim(s[6]) == "a b c");
        test_(trim(s[7]) == "a b c");
        test_(trim(s[8]) == "a \t b \t c");
        test_(trim(s[9]) == "");
        test_(trim(s[10]) == "");
    }
    void run() {
        testTrim();
    }
};
#endif // TRIMTEST_H ///:~

//: C03:TrimTest.cpp {0}
#include "TrimTest.h"

// Initialize static data
std::string TrimTest::s[TrimTest::NTESTS] = {
    " \t abcdefghijklmno \t ",
    "abcdefghijklmno \t ",
    " \t abcdefghijklmno",
    "a", "ab", "abc", "a b c",
    " \t a b c \t ", " \t a \t b \t c \t ",
    "\t \n \r \v \f",
    "" // Must also test the empty string
}; ///:~

//: C03:TrimTestMain.cpp
//{L} ../TestSuite/Test TrimTest
#include "TrimTest.h"

int main() {
    TrimTest t;
    t.run();
    return t.report();
} ///:~
```

读者可以看到，在**strings**型数组中字符型数组自动转换成了**string**对象。读者可以使用这个数组提供测试案例，检查**string**两端的空格和制表符是否删除了，以及确定**string**中间的空格和制表符是否保留了下来。

### 3.4.3 从字符串中删除字符

使用**erase()**成员函数删除字符串中的字符是简单而有效的。这个函数有两个参数：一个参数表示开始删除字符的位置（默认值是0）；另一个参数表示要删除多少个字符（默认值是**string::npos**）。如果指定删除的字符个数比字符串中剩余的字符还多，那么剩余的字符将全部被删除（所以调用不含参数的**erase()**函数将删除字符串中的所有字符）。有时，删除一个HTML文件中的标记（tag）与特殊字符是很有用的，这样就可以得到类似于浏览器中所显示的文本文件，仅仅作作为纯文本文件。下面这个例子用**erase()**来完成这个工作：

```
//: C03:HTMLStripper.cpp {RunByHand}
//{L} ReplaceAll
// Filter to remove html tags and markers.
#include <cassert>
#include <cmath>
#include <cstddef>
#include <fstream>
#include <iostream>
#include <string>
#include "ReplaceAll.h"
#include "../require.h"
using namespace std;

string& stripHTMLTags(string& s) {
    static bool inTag = false;
    bool done = false;
    while(!done) {
        if(inTag) {
            // The previous line started an HTML tag
            // but didn't finish. Must search for '>'.
            size_t rightPos = s.find('>');
            if(rightPos != string::npos) {
                inTag = false;
                s.erase(0, rightPos + 1);
            }
            else {
                done = true;
                s.erase();
            }
        }
        else {
            // Look for start of tag:
            size_t leftPos = s.find('<');
            if(leftPos != string::npos) {
                // See if tag close is in this line:
                size_t rightPos = s.find('>');
                if(rightPos == string::npos) {
                    inTag = done = true;
                    s.erase(leftPos);
                }
                else
                    s.erase(leftPos, rightPos - leftPos + 1);
            }
            else
                done = true;
        }
    }
    // Remove all special HTML characters
    replaceAll(s, "&lt;", "<");
    replaceAll(s, "&gt;", ">");
    replaceAll(s, "&amp;", "&");
}
```



```

    replaceAll(s, "&nbsp;", " ");
    // Etc...
    return s;
}

int main(int argc, char* argv[]) {
    requireArgs(argc, 1,
        "usage: HTMLStripper InputFile");
    ifstream in(argv[1]);
    assure(in, argv[1]);
    string s;
    while(getline(in, s))
        if(!stripHTMLTags(s).empty())
            cout << s << endl;
} ///:~

```

这个例子甚至能够删除跨越多行<sup>①</sup>的HTML标记。这归功于静态标志**inTag**，一旦发现了开始标记，此逻辑标志就会被设为**true**，无论相应的结束标记是否与这个开始标记在同一行。所有形式的**erase()**都包括在**stripHTMLFlags()**函数中。<sup>②</sup>在这里所用的**getline()**版本是在**<string>**头文件中声明的（全局）函数，这个函数使用起来很方便，因为它可以在其**string**参数中存储任意长度的一行文本。在使用**istream::getline()**时不必考虑所用字符序列（array）维数的大小。注意，此程序使用了本章开始时介绍的**replaceAll()**函数。下一章将采用字符串流来构造一个更加优秀的解法。

#### 3.4.4 字符串的比较

字符串的比较与数字的比较有其固有的不同。数字有恒定的永远有意义的值。为了评定两个字符串的大小关系，必须进行字典比较（lexical comparison）。字典比较的意思是，当测试一个字符看它是“大于”还是“小于”另一个字符时，实际比较的是它们的数值表示，而这些数值表示是由当前所使用的字符集的校对序列来决定的。通常，这种校对序列是ASCII校对序列，它给英语的可打印字符分配的数值为从32到127范围内的连续十进制数字。在ASCII校对序列中，序列表中第一个“字符”是空格，然后是几个常用标点符号，再往后是大小写字母。遵照字母表的编排，比较靠前的字符的ASCII码值都低于比较靠后的字符。知道了这些细节，了解和记忆以下事实就更容易了：当字典比较报告字符串**s1**“大于”字符串**s2**时，也即两者相比较时遇到第1对不同的字符时，字符串**s1**中第1个不同的字符比字符串**s2**中同样位置的字符在ASCII表中的位置更靠后。

C++提供了多种字符串比较方法，它们各具特色。其中最简单的就是使用非成员的重载运算符函数：**operator ==**、**operator !=**、**operator >**、**operator <**、**operator >=**和**operator <=**。

```

//: C03:CompStr.h
#ifndef COMPSTR_H
#define COMPSTR_H
#include <string>
#include "../TestSuite/Test.h"
using std::string;

class CompStrTest : public TestSuite::Test {
public:

```

① 为了简化说明，这一版本不考虑嵌套标记，例如注释。

② 使用数学方法来引发一些对**erase()**的调用，在此是很有吸引力的。由于某些情况下其操作数之一是**string::npos**（可能得到的最大无符号整型变量），整型溢出就可能发生，进而会搞垮整个算法。

```

void run() {
    // Strings to compare
    string s1("This");
    string s2("That");
    test_(s1 == s1);
    test_(s1 != s2);
    test_(s1 > s2);
    test_(s1 >= s2);
    test_(s1 >= s1);
    test_(s2 < s1);
    test_(s2 <= s1);
    test_(s1 <= s1);
}
};
#endif // COMPSTR_H ///:~

//: C03:CompStr.cpp
//{L} ../TestSuite/Test
#include "CompStr.h"

int main() {
    CompStrTest t;
    t.run();
    return t.report();
} ///:~

```

重载的比较运算符不但能进行字符串全串比较还能进行字符串的个别字符元素的比较。

在下面的例子中，注意在比较运算符左右两边的自变量类型的灵活性。为了高效率地运行，对于字符串对象、引用文字和指向C语言风格的字符串的指针等的直接比较，**string**类不创建临时**string**对象，而是采用重载运算符进行。

```

//: C03:Equivalence.cpp
#include <iostream>
#include <string>
using namespace std;

int main() {
    string s2("That"), s1("This");
    // The lvalue is a quoted literal
    // and the rvalue is a string:
    if("That" == s2)
        cout << "A match" << endl;
    // The left operand is a string and the right is
    // a pointer to a C-style null terminated string:
    if(s1 != s2.c_str())
        cout << "No match" << endl;
} ///:~

```

**c\_str()**函数返回一个**const char\***，它指向一个C语言风格的具有“空结束符”的字符串，此字符串与**string**对象的内容等价。当想将一个字符串传送给一个标准C语言函数时，比如**atoi()**或**<cstring>**头文件中定义的任一函数，**const char\***可派得上用场。不过，将**c\_str()**的返回值作为非**const**参数应用于任一函数都是错误的。

在字符串的运算符中，不会找到逻辑非(!)或逻辑比较运算符(&&和||)。(也不会找到重载版的C语言逐位(二进制数位)运算符&、|、^或~。)重载字符串类的非成员比较运算符被限定在一个可以清晰地、无二义性地应用于多个字符或字符组的子集中。

**compare()**成员函数能够提供远比非成员运算符集更复杂精密的比较手段。它提供的那些重载版本，可以比较：

- 两个完整的字符串。
- 一个字符串的某一部分与另一字符串的全部。
- 两个字符串的子集。

下面的例子用来比较两个完整的字符串：

```
//: C03:Compare.cpp
// Demonstrates compare() and swap().
#include <cassert>
#include <string>
using namespace std;

int main() {
    string first("This");
    string second("That");
    assert(first.compare(first) == 0);
    assert(second.compare(second) == 0);
    // Which is lexically greater?
    assert(first.compare(second) > 0);
    assert(second.compare(first) < 0);
    first.swap(second);
    assert(first.compare(second) < 0);
    assert(second.compare(first) > 0);
} ///:~
```

本例中**swap()**函数所做的工作，顾名思义是：交换其自身对象和参数的内容。为了对一个字符串或两个字符串中的字符子集进行比较，可加上两个参数，一个参数定义开始比较的位置，另一个参数定义字符子集要考虑的字符个数。例如，可以使用下面这个**compare()**函数的重载版：

```
s1.compare(s1StartPos, s1NumberChars, s2, s2StartPos,  
s2NumberChars);
```

举例如下：

```
//: C03:Compare2.cpp
// Illustrate overloaded compare().
#include <cassert>
#include <string>
using namespace std;

int main() {
    string first("This is a day that will live in infamy");
    string second("I don't believe that this is what "
        "I signed up for");
    // Compare "his is" in both strings:
    assert(first.compare(1, 7, second, 22, 7) == 0);
    // Compare "his is a" to "his is w":
    assert(first.compare(1, 9, second, 22, 9) < 0);
} ///:~
```

在以往的例子中，如果涉及字符串中的个别字符，教材中都使用C语言风格的数组索引语法。C++中的字符串类提供一种**s[n]**表示法的替代方法：**at()**成员函数。如果不出现意外事件，在C++中这两种索引机制产生的结果是一样的：

```
//: C03:StringIndexing.cpp
#include <cassert>
#include <string>
using namespace std;
```

```
int main() {
    string s("1234");
    assert(s[1] == '2');
    assert(s.at(1) == '2');
} ///:~
```

然而，数组索引下标表示`[ ]`与`at( )`之间有一个重要的不同点。如果程序员想引用一个超过边界的数组元素，`at( )`将会友好地抛出一个异常，而普通的`[ ]`下标语法将让程序员自行决策：

```
///: C03:BadStringIndexing.cpp
#include <exception>
#include <iostream>
#include <string>
using namespace std;

int main() {
    string s("1234");
    // at() saves you by throwing an exception:
    try {
        s.at(5);
    } catch(exception& e) {
        cerr << e.what() << endl;
    }
} ///:~
```

有责任心的程序员不会去用有冒险性的索引，程序员希望能够从自动边界检查中受益。使用`at( )`代替`[ ]`，就有机会从容地修复由于引用了不存在的数组元素而产生的错误。在一个测试编译器上执行这个程序，得到的输出结果是：

```
invalid string position
```

`at( )`成员抛出的是一个`out_of_range`类对象，它（最终）派生于`std::exception`。程序可在一个异常处理器中捕获该对象，并采取适当的补救措施，比如重新计算越界下标或扩充数组。采用`string::operator[ ]( )`不会有那样的保护性，它的危险性等同于C语言中对`char`型数组的处理。<sup>①</sup>

### 3.4.5 字符串和字符的特性

本章前面的程序`Find.cpp`可能导致读者提出下面这个显而易见的问题：为什么对大小写不敏感的比较没有成为标准`string`类的一部分？对此问题的回答揭示了关于C++字符串对象真实性质的有趣背景。

读者可以考虑一下，字符有“大小写”到底意味着什么。希伯来语、波斯语和日本汉字并不使用大小写的概念，即对这些语言来说大小写没有什么意义。这似乎是说，如果有方法将一些语言指定为“全大写”或“全小写”，就能够设计出通用的解决方案。但是，某些采用“大小写”概念的语言，同时也用可区别的标记改变了特殊字符的意义，如：西班牙语中的变音符号，法语中的抑扬符号，还有德语中的元音变音。因此，任何试图全面解决此问题的大小写敏感的分类整理方案，最终都会变得非常复杂直至不能进行下去。

虽然通常将C++ `string`看成一个类，但事实并非如此。需要说明一下，`basic_string< >`模板是一种更通用的工具，而`string`类型只是其更专门化的版本。请看`string`在标准C++头文

① 鉴于上述安全原因，C++标准制定委员会正考虑一个议案来对`string::operator[ ]`进行重新定义，以便在C++0x中使其与`string::at( )`等价。



件里的声明：<sup>①</sup>

```
typedef basic_string<char> string;
```

要了解字符串类的本质，请看**basic\_string<>**模板：

```
template<class charT, class traits = char_traits<charT>,
        class allocator = allocator<charT> > class basic_string;
```

本教材将在第5章中详细讨论模板（比第1卷第16章要详细得多）。但现在，只需注意一下**string**类型是通过使用**char**实例化**basic\_string**模板而创建的。在**basic\_string<>**模板声明内部，下面的一行：

```
class traits = char_traits<charT>.
```

告诉读者基于**basic\_string<>**模板的类的行为，是由基于**char\_traits<>**模板的某个类指定的。因此，**basic\_string<>**模板产生的是面向字符串的类，此类的操作对象是除了**char**以外的类型（比如宽字符（wide character））。为了达到这一目的，**char\_traits<>**模板控制多种字符集的内容和校对行为，而这些字符集用的是字符比较函数**eq()**（相等），**ne()**（不等）和**lt()**（小于）。**basic\_string<>**字符串的比较函数就依赖于这些函数。

这就是为什么字符串类不包含对大小写不敏感的成员函数的原因：因为那不属于它的本职工作。为了改变字符串类比较字符的方式，必须提供不同的**char\_traits<>**模板，因为它定义了对个别字符进行比较的成员函数的行为。

可以用此信息构造一种忽略大小写的新类型的**string**类。首先，定义一个从现存模板中继承的一种对大小写不敏感的新的**char\_traits<>**模板。其次，仅重写需要更改的成员，使其能逐个字符进行大小写不敏感比较（除了之前提及的3个对字符进行词典比较的成员函数之外，还会为**char\_traits**提供函数**find()**和**compare()**的新的实现）。最后，我们将用**typedef**定义一个基于**basic\_string**的新类，但使用对大小写不敏感的**ichar\_traits**模板作为第2个参数：

```
//: C03:ichar_traits.h
// Creating your own character traits.
#ifndef ICHAR_TRAITS_H
#define ICHAR_TRAITS_H
#include <cassert>
#include <cctype>
#include <cmath>
#include <cstddef>
#include <ostream>
#include <string>
using std::allocator;
using std::basic_string;
using std::char_traits;
using std::ostream;
using std::size_t;
using std::string;
using std::toupper;
using std::tolower;

struct ichar_traits : char_traits<char> {
    // We'll only change character-by-
```

① 读者实现时可定义这里的所有3个模板参数。由于最后两个模板参数有默认值，那样一个声明与在此写的内容是等价的。

```

// character comparison functions
static bool eq(char c1st, char c2nd) {
    return toupper(c1st) == toupper(c2nd);
}
static bool ne(char c1st, char c2nd) {
    return !eq(c1st, c2nd);
}
static bool lt(char c1st, char c2nd) {
    return toupper(c1st) < toupper(c2nd);
}
static int
compare(const char* str1, const char* str2, size_t n) {
    for(size_t i = 0; i < n; ++i) {
        if(str1 == 0)
            return -1;
        else if(str2 == 0)
            return 1;
        else if(tolower(*str1) < tolower(*str2))
            return -1;
        else if(tolower(*str1) > tolower(*str2))
            return 1;
        assert(tolower(*str1) == tolower(*str2));
        ++str1; ++str2; // Compare the other chars
    }
    return 0;
}
static const char*
find(const char* s1, size_t n, char c) {
    while(n-- > 0)
        if(toupper(*s1) == toupper(c))
            return s1;
        else
            ++s1;
    return 0;
}
};

typedef basic_string<char, ichar_traits> istring;

inline ostream& operator<<(ostream& os, const istring& s) {
    return os << string(s.c_str(), s.length());
}
#endif // ICHAR_TRAITS_H ///:~

```

该程序提供了一个**typedef**命名的**istring**类，这样该类就能在各方面像普通的**string**类一样工作，除了在进行比较的时候不考虑大小写。为了方便起见，程序也提供了一种重载的**operator <<()**，以便打印**istring**。举例如下：

```

//: C03:ICompare.cpp
#include <cassert>
#include <iostream>
#include "ichar_traits.h"
using namespace std;

int main() {
    // The same letters except for case:
    istring first = "tHis";
    istring second = "ThIS";
    cout << first << endl;
    cout << second << endl;
    assert(first.compare(second) == 0);
}

```

```

    assert(first.find('h') == 1);
    assert(first.find('I') == 2);
    assert(first.find('x') == string::npos);
} ///:~

```

它只是一个很小的也没有什么实用价值的例子。为使**istring**完全等价于**string**，还得创建其他必要的函数以便支持新的**istring**类型。

通过下面的**typedef**，**<string>** 头文件提供宽字符串类：

```
typedef basic_string<wchar_t> wstring;
```

在宽字符流（wide stream）（代替**ostream**的**wostream**，也在**<iostream>** 中定义）和头文件**<cwctype>**（**<cctype>** 的宽字符版本）中，也体现出对宽字符串的支持。运用这些，再加上标准库里**char\_traits**中的**wchar\_t**说明，就可以完成**ichar\_traits**的宽字符版本：

```

//: C03: iwchar_traits.h {-g++}
// Creating your own wide-character traits.
#ifndef IWCHAR_TRAITS_H
#define IWCHAR_TRAITS_H
#include <cassert>
#include <cmath>
#include <cstddef>
#include <cwctype>
#include <ostream>
#include <string>

using std::allocator;
using std::basic_string;
using std::char_traits;
using std::size_t;
using std::tolower;
using std::toupper;
using std::wostream;
using std::wstring;

struct iwchar_traits : char_traits<wchar_t> {
    // We'll only change character-by-
    // character comparison functions
    static bool eq(wchar_t c1st, wchar_t c2nd) {
        return towupper(c1st) == towupper(c2nd);
    }
    static bool ne(wchar_t c1st, wchar_t c2nd) {
        return towupper(c1st) != towupper(c2nd);
    }
    static bool lt(wchar_t c1st, wchar_t c2nd) {
        return towupper(c1st) < towupper(c2nd);
    }
    static int compare(
        const wchar_t* str1, const wchar_t* str2, size_t n) {
        for(size_t i = 0; i < n; i++) {
            if(str1 == 0)
                return -1;
            else if(str2 == 0)
                return 1;
            else if(towlower(*str1) < tolower(*str2))
                return -1;
            else if(towlower(*str1) > tolower(*str2))
                return 1;
            assert(towlower(*str1) == tolower(*str2));
            ++str1; ++str2; // Compare the other wchar_ts
        }
    }
}

```



```

        return 0;
    }
    static const wchar_t*
    find(const wchar_t* s1, size_t n, wchar_t c) {
        while(n-- > 0)
            if(towupper(*s1) == towupper(c))
                return s1;
            else
                ++s1;
        return 0;
    }
};

typedef basic_string<wchar_t, iwchar_traits> iwstring;

inline wostream& operator<<(wostream& os,
    const iwstring& s) {
    return os << wstring(s.c_str(), s.length());
}
#endif // IWCHAR_TRAITS_H ///:~

```

如同读者所见，这基本上是一个要求在源代码中的适当位置放置一个‘w’的练习。测试程序如下所示：

```

//: C03:IWCompare.cpp {-g++}
#include <cassert>
#include <iostream>
#include "iwchar_traits.h"
using namespace std;

int main() {
    // The same letters except for case:
    iwstring wfirst = L"tHis";
    iwstring wsecond = L"ThIS";
    wcout << wfirst << endl;
    wcout << wsecond << endl;
    assert(wfirst.compare(wsecond) == 0);
    assert(wfirst.find('h') == 1);
    assert(wfirst.find('I') == 2);
    assert(wfirst.find('x') == wstring::npos);
} ///:~

```

遗憾的是，某些编译器对宽字符仍然没有提供足够的支持。

### 3.5 字符串的应用

如果仔细查看本书的程序举例代码，读者会注意到注释中的一些标记。这些都是供Bruce所编写的一个Python程序使用的，该程序用于将代码提取到文件并生成makefile的程序。例如，以双斜线加冒号开始的一行表示源文件的第1行。其余行所描述的信息包括文件名、文件所在的位置以及是否应该只编译文件，而不是生成可执行文件。例如，在上面的程序中，第1行包含字符串**C03:IWCompare.cpp**，它表示文件**IWCompare.cpp**应该被提取到目录**C03**下。

源文件的最后一行包括3条斜线，其后是一个冒号和一个波形号。如果第1行有一个惊叹号，并且其后紧接着冒号，源代码的第1行和最后一行就不会输出到文件上（这只适用于数据文件）。（为什么在代码中隐藏这些标记呢，那是因为当将代码提取器应用于本教材的代码正文时，我们不想破坏提取器。）

Bruce的Python程序所做的远不止提取代码。如果文件名后有标记“{O}”，它在makefile中的条目将被设置为只编译文件，而不将其链接到可执行文件中。（第2章的测试框架就是这么构建的。）为了将这样一个文件与另一个源代码例子链接起来，目标执行文件的源文件会包括一个“{L}”指令，就像：

```
//{L} ../TestSuite/Test
```

本部分将介绍一个程序，该程序仅用来提取所有的代码，以便程序员进行手工编译和检查。程序员可以用这个程序来提取本教材中的所有代码，并将文档保存为文本文件<sup>①</sup>（称其为TICV2.txt），在shell命令行中执行类似下面的命令：

```
C:> extractCode TICV2.txt /TheCode
```

这个命令读取文本文件**TICV2.txt**，然后在根目录/**TheCode**下的子目录里写出所有源代码文件。目录树如下所示：

```
TheCode/
  C0B/
  C01/
  C02/
  C03/
  C04/
  C05/
  C06/
  C07/
  C08/
  C09/
  C10/
  C11/
  TestSuite/
```

各章中例子的源文件包括在相应的目录里。

下面是程序：

```
//: C03:ExtractCode.cpp {-edg} {RunByHand}
// Extracts code from text.
#include <cassert>
#include <cstdint>
#include <cstdio>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
// Legacy non-standard C header for mkdir()
#if defined(__GNUC__) || defined(__MWERKS__)
#include <sys/stat.h>
#elif defined(__BORLANDC__) || defined(_MSC_VER) \
    || defined(__DMC__)
#include <direct.h>
#else
#error Compiler not supported
#endif

// Check to see if directory exists
```

① 注意，当文件被存成文本时，Microsoft Word的某些版本将会错误地将单个引述字符替换成扩展了的ASCII码字符，这会造成编译错误。我们不知道错误产生的原因。读者只需用单引号手工替换那个字符就行了。

```

// by attempting to open a new file
// for output within it.
bool exists(string fname) {
    size_t len = fname.length();
    if(fname[len-1] != '/' && fname[len-1] != '\\')
        fname.append("/");
    fname.append("000.tmp");
    ofstream outf(fname.c_str());
    bool existFlag = outf;
    if(outf) {
        outf.close();
        remove(fname.c_str());
    }
    return existFlag;
}

int main(int argc, char* argv[]) {
    // See if input file name provided
    if(argc == 1) {
        cerr << "usage: extractCode file [dir]" << endl;
        exit(EXIT_FAILURE);
    }
    // See if input file exists
    ifstream inf(argv[1]);
    if(!inf) {
        cerr << "error opening file: " << argv[1] << endl;
        exit(EXIT_FAILURE);
    }
    // Check for optional output directory
    string root("./"); // current is default
    if(argc == 3) {
        // See if output directory exists
        root = argv[2];
        if(!exists(root)) {
            cerr << "no such directory: " << root << endl;
            exit(EXIT_FAILURE);
        }
        size_t rootLen = root.length();
        if(root[rootLen-1] != '/' && root[rootLen-1] != '\\')
            root.append("/");
    }
    // Read input file line by line
    // checking for code delimiters
    string line;
    bool inCode = false;
    bool printDelims = true;
    ofstream outf;
    while(getline(inf, line)) {
        size_t findDelim = line.find("//" "/*:~");
        if(findDelim != string::npos) {
            // Output last line and close file
            if(!inCode) {
                cerr << "Lines out of order" << endl;
                exit(EXIT_FAILURE);
            }
            assert(outf);
            if(printDelims)
                outf << line << endl;
            outf.close();
            inCode = false;
            printDelims = true;
        } else {
            findDelim = line.find("//" "/*:");

```



```

if(findDelim == 0) {
    // Check for '!' directive
    if(line[3] == '!') {
        printDelims = false;
        ++findDelim; // To skip '!' for next search
    }
    // Extract subdirectory name, if any
    size_t startOfSubdir =
        line.find_first_not_of(" \t", findDelim+3);
    findDelim = line.find(':', startOfSubdir);
    if(findDelim == string::npos) {
        cerr << "missing filename information\n" << endl;
        exit(EXIT_FAILURE);
    }
    string subdir;
    if(findDelim > startOfSubdir)
        subdir = line.substr(startOfSubdir,
                               findDelim - startOfSubdir);
    // Extract file name (better be one!)
    size_t startOfFile = findDelim + 1;
    size_t endOfFile =
        line.find_first_of(" \t", startOfFile);
    if(endOfFile == startOfFile) {
        cerr << "missing filename" << endl;
        exit(EXIT_FAILURE);
    }
    // We have all the pieces; build fullPath name
    string fullPath(root);
    if(subdir.length() > 0)
        fullPath.append(subdir).append("/");
    assert(fullPath[fullPath.length()-1] == '/');
    if(!exists(fullPath))
#if defined(__GNUC__) || defined(__MWERKS__)
        mkdir(fullPath.c_str(), 0); // Create subdir
#else
        mkdir(fullPath.c_str()); // Create subdir
#endif
    fullPath.append(line.substr(startOfFile,
                                endOfFile - startOfFile));
    outf.open(fullPath.c_str());
    if(!outf) {
        cerr << "error opening " << fullPath
              << " for output" << endl;
        exit(EXIT_FAILURE);
    }
    inCode = true;
    cout << "Processing " << fullPath << endl;
    if(printDelims)
        outf << line << endl;
}
else if(inCode) {
    assert(outf);
    outf << line << endl; // Output middle code line
}
}
}
exit(EXIT_SUCCESS);
} ///:~

```

首先,读者应注意某些条件编译指令。用于在文件系统中创建目录的**mkdir()**函数,是

由 POSIX 标准<sup>①</sup>在头文件 `<sys/stat.h>` 中定义的。遗憾的是，很多编译器仍然在使用不同的另一个头文件 (`<direct.h>`)。对于不同的头文件，各自的 `mkdir()` 识别标志也有所不同：POSIX 指定了两个参数，而旧版本只有一个。因此，在以后的程序中就有了更多的条件编译，以便选择正确的 `mkdir()` 进行调用。在本教材的例子中通常不使用条件编译，但是这个特别的程序太有用了，以至于不能放置哪怕一点额外的工作进去，因为读者能用它提取教材中所有的代码。

`ExtractCode.cpp` 中的 `exists()` 函数通过打开目录中一个临时文件的方式来判断这个目录是否存在。如果打开文件失败，目录就不存在。要删除一个文件，可以将其 `char*` 型名字传送到 `std::remove()` 中。

主程序首先判定命令行参数的合法性，然后一次一行地读取输入文件，同时查找特殊的源代码定界符。布尔标志符 `inCode` 表示程序在源文件的中间，所以这些代码行应当输出。如果源代码的开始标记不是跟随惊叹号，`printDelims` 标志符将为真；否则第1行与最后一行将不会写出。首先查看结束定界符的存在性，这一点很重要，因为开始标记是结束定界符的一个子集。如果先查找开始标记，则程序在找到开始标记和结束定界符的时候都会返回成功。如果遇到结束标记，程序就知道源文件正在处理过程中；否则，文本文件中定界符的摆放方式就有错误了。如果 `inCode` 为真，那就没什么问题，程序（可选地）写下最后一行然后关闭文件。当找到开始标记时，系统就从语法上分析目录和文件名的组成，然后打开文件。下面几个与 `string` 有关的函数在此例中都用到了：`length()`、`append()`、`getline()`、`find()`（两个版本）、`find_first_not_of()`、`substr()`、`find_first_of()`、`c_str()`，当然还有 `operator <<()`。

### 3.6 小结

C++ `string` 对象的优越性是 C 语言中相关功能难以望其项背的，这给程序研发者带来了极大的便利。在很大程度上，`string` 类使得通过字符型指针来引用字符串已经不再必要了。这就从根本上消除了由于使用未经初始化的指针或具有不正确值的指针造成的一系列软件缺陷。

为了适应字符串中数据长度增长变化的需要，C++ 字符串动态且透明地扩充其内部的数据存储空间。当字符串中存储的数据增长超过最初分配给它的内存空间边界时，字符串对象就会进行存储管理调用，从堆中提取和归还存储空间。稳定的存储分配方案避免了内存泄漏，并且有可能比“依靠（编程人员）自己转来转去”的内存管理方式更加有效。

`string` 类成员函数为字符串的创建、修改和查找提供了相当广泛的工具集。字符串的比较总是大小写敏感的，但也可对字符串进行大小写不敏感的比较。方法是先将字符串数据复制到具有 C 语言风格的带空结束符的字符串中，然后调用大小写不敏感字符串比较函数，暂时将字符串对象中存放的数据转换成单一的大写或小写字母；也可以创建大小写不敏感的字符串类，重载用来创建 `basic_string` 对象的字符特性。

### 3.7 练习

3-1 编写并测试一个函数，逆转字符串中字符的顺序。

3-2 回文是一个单词或词组，不管从前还是从后开始读，结果都是一样的。例如“madam”或“wow”。编写一个程序，接受来自命令行的一个字符串参数，使用在上一个练习

① POSIX 是一个 IEEE 标准，支持“便携式操作系统接口（Portable Operating System Interface）”在 Unix 系统中的许多低级系统调用，POSIX 就是这些调用的一体化产物。



(3-1) 中编写的函数, 打印出这个字符串是否为回文。

3-3 修改在练习3-2中编写的程序, 如果位置对称的两个字母大小写不同, 仍然使其返回 **true**。例如 “Civic” 仍会返回 **true**, 虽然第一个字母是大写字母。

3-4 修改练习3-3中的程序, 使其能够忽略标点符号与空格。例如 “Able was I, ere I saw Elba.” 也报告 **true**。

3-5 使用下面字符串声明并且只能用 **char** (不能用印刷错误的串或不可思议的数字):

```
string one("I walked down the canyon with the moving
mountain bikers.");
string two("The bikers passed by me too close for
comfort.");
string three("I went hiking instead.");
```

生成下列句子:

```
I moved down the canyon with the mountain bikers. The
mountain bikers passed by me too close for comfort. So
I went hiking instead.
```

3-6 编写一个名为 **replace** 的程序, 接受3个命令行参数, 其中一个参数表示输入的文本文件; 一个参数表示被替换掉的字符串 (称为 **from**); 还有一个表示替换后的字符串 (称为 **to**)。此程序应该将一个新文件写到标准输出, 并将所有的 **from** 被 **to** 代替的事件显示出来。

3-7 重写练习3-6, 忽略大小写, 替换所有 **from**。

3-8 使练习3-3中的程序获得一个来自命令行的文件名, 然后显示此文件中所有是回文的单词 (忽略大小写)。不要重复显示 (即使它们的大小写不同)。所找的回文仅限于单词。(与练习3-4不同。)

3-9 修改 **HTMLStripper.cpp**, 使其在遇到一个标记时就显示这个标记的名字。然后还显示在这个标记与相应的结束标记之间的内容。假设无标记的嵌套, 并且所有的标记都有结束标记 (表示为 **<TAGNAME>**)。

3-10 编写一个程序, 采用3个命令行参数 (一个文件名和两个字符串)。按照程序开头处的用户输入 (用户会选择使用那一种匹配模式), 将文件中那些含有两个字符串的行, 两个字符串中任意一个字符串的行、只有一个字符串的行、或两个字符串都不含的行, 全部显示到屏幕。除了 “两个字符串都不含” 的情况外, 为了突出强调输入的字符串, 在每一个显示的字符串的开头与结尾全部标上星号 (\*)。

3-11 编写一个程序, 采用两个命令行参数 (一个文件名和一个字符串), 计算字符串在文件中出现的次数, 包括其作为子串出现的情况 (但不计重叠)。例如, 输入字符串 “ba” 将在单词 “basketball” 中匹配两次, 但输入字符串 “ana” 在单词 “banana” 中只匹配一次。将字符串在文件中匹配的次数, 还有出现字符串的单词的平均长度显示到屏幕。(如果字符串在单词中出现的次数大于1, 当计算该单词的平均长度时, 只将该单词计算一次。)

3-12 编写一个程序, 使用来自命令行的一个文件名, 并对字符使用情况进行统计, 包括标点符号与空格 (所有的字符值是从 **0x21[33]** 到 **0x7E[126]**, 还包括空格字符)。也就是说, 计算每个字符在文件中出现的次数, 然后将它们按 **ASCII** 排列顺序 (空格, 然后 !, ", #, 等等), 或按用户在程序开始时输入的字符使用频率的升序或降序来显示其结果。对于空格, 显示单词 “Space” 而非单个空字符 ' '。程序运行结果如下所示:

```

Format sequentially, ascending, or descending
(S/A/D): D
t: 526
r: 490
etc.

```

- 3-13 使用**find( )**和**rfind( )**，编写一个程序。它采用两个命令行参数（一个文件名和一个字符串），显示与该字符串不匹配的第1个和最后一个单词（包括它们的索引值），还有该字符串出现的第一个与最后一个的索引值。当上述任一查找都失败时，显示“Not Found”。
- 3-14 使用**find\_first\_of**函数“族”（但不是惟一的）。编写一个程序，删掉文件中所有非字母数字型的字符（空格与句号除外），然后将句号后的第1个字母大写。
- 3-15 再次使用**find\_first\_of**函数“族”。编写一个程序，将一个文件名用作命令行参数，然后将文件中所有的数字格式化为货币值。忽略第1个十进制小数点与其后第1个非数值型字符之间的所有小数点，将所得数值四舍五入到百分位。例如，字符串 12.399abc 29.00.6a（美式转换）将被格式化为 \$12.40 abc\$ 29.01a。
- 3-16 编写一个程序，采用两个命令行参数（一个文件名和一个数字），搅乱文件中的每一个单词：随机交换每个单词中的两个字母，交换次数由第2个参数提供。（即，如果从命令行传送到程序中的是0，就不能搅乱单词；如果传送进来的是1，一对随机选择的字母应被交换；如果输入的是2，两对随机选择字母将被交换；以此类推。）
- 3-17 编写一个程序，从命令行获得一个文件名，显示其中句子的个数（定义为文件中句号的个数）、每个句子中字符的平均个数，还有文件中字符的总个数。
- 3-18 自行证明，当有越界情况发生时，**at( )**成员函数确实会抛出一个异常，但是索引运算符**[ ]**则不会这么做。



## 输入输出流

处理一般的I/O问题，比仅仅使用标准I/O库函数并把它变成一个类需要做更多的工作。

如果能把所有平常的“容器 (receptacle)”——标准I/O函数、文件以及内存块——看做相同的对象，都使用相同的接口进行操作，这不是很好吗？这种思想是建立在输入输出流之上的。与C语言**stdio**（标准输入/输出）库中各式各样的函数相比，输入输出流使用起来更容易、更安全，有时甚至更高效。

C++类库中的输入输出流类通常是C++初学者最先学习使用的部分。本章讨论输入输出流中比C语言中**stdio**更强大的功能，阐述了文件流、字符串流和标准控制台流。

### 4.1 为什么引入输入输出流

读者可能想知道以前的C库到底有什么不好。为什么不把C库封装成新的类呢？有时这是一种好的解决办法。例如，**stdio**中定义的**FILE**为指向文件的指针，假定现在需要安全地打开文件并且不依赖用户调用**close()**来关闭它，下面的程序可以实现这一目标：

```
//: C04:FileClass.h
// stdio files wrapped.
#ifndef FILECLASS_H
#define FILECLASS_H
#include <cstdio>
#include <stdexcept>

class FileClass {
    std::FILE* f;
public:
    struct FileClassError : std::runtime_error {
        FileClassError(const char* msg)
            : std::runtime_error(msg) {}
    };
    FileClass(const char* fname, const char* mode = "r");
    ~FileClass();
    std::FILE* fp();
};
#endif // FILECLASS_H ///:~
```

当在C语言中进行文件I/O时，是使用无保护的指向**FILE struct**的指针来完成有关操作，但这个类封装了文件结构指针，并且用构造函数和析构函数来确保指针被正确地初始化和清理。构造函数的第2个参数是文件打开模式，默认值为“**r**”即“只读模式”。

为了在文件I/O函数中使用这个指针的值，可以用存取访问函数（access function）**fp()**取得它。下面是这个成员函数的定义：

```
//: C04:FileClass.cpp {0}
// FileClass Implementation.
#include "FileClass.h"
#include <cstdlib>
#include <cstdio>
using namespace std;

FileClass::FileClass(const char* fname, const char* mode) {
```

```

        if((f = fopen(fname, mode)) == 0)
            throw FileClassError("Error opening file");
    }

    FileClass::~FileClass() { fclose(f); }

    FILE* FileClass::fp() { return f; } ///:~

```

就像平常所做的一样，构造函数调用**fopen()**，而且要确保返回结果不为零，结果为零说明打开文件失败。如果文件不能正常打开，则抛出异常。

析构函数用来关闭文件，而存取访问函数**fp()**则返回指针**f**。下面是使用**FileClass**的一个简单例子：

```

//: C04:FileClassTest.cpp
//{L} FileClass
#include <cstdlib>
#include <iostream>
#include "FileClass.h"
using namespace std;
int main() {
    try {
        FileClass f("FileClassTest.cpp");
        const int BSIZE = 100;
        char buf[BSIZE];
        while(fgets(buf, BSIZE, f.fp()))
            fputs(buf, stdout);
    } catch(FileClass::FileClassError& e) {
        cout << e.what() << endl;
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
} // File automatically closed by destructor
///:~

```

现在，创建一个**FileClass**对象并在普通的C文件I/O函数中通过调用**fp()**使用它。当用完这个对象之后就不需要再理会它了；当文件对象超出其作用域后，析构函数会关闭该文件。

虽然**FILE**指针是私有的，但它并不是特别安全，因为成员函数**fp()**可以检索它。既然惟一的作用似乎只是为了确保指针能被正确初始化和清除，那么为什么不把它设计成公有的或使用**struct**来代替呢？注意，当能够用函数**fp()**取得指针**f**的一个拷贝的时候，不能同时给**f**赋值——这项操作完全由类来控制。得到由**fp()**返回的指针后，客户程序员仍然能给结构元素赋值或对其进行进一步处理，所以从安全的角度对于**FILE**指针，与其确保其合法性还不如将其作为结构的固有成员。

如果需要得到完全的安全，就必须防止客户直接存取**FILE**指针。所有的常用文件I/O函数都必须作为成员函数封装在类中，使得借助于C语言能做到的每一件事，在C++类中均可做到：

```

//: C04:Fullwrap.h
// Completely hidden file IO.
#ifndef FULLWRAP_H
#define FULLWRAP_H
#include <cstddef>
#include <cstdio>
#undef getc
#undef putc
#undef ungetc
using std::size_t;
using std::fpos_t;

```

```

class File {
    std::FILE* f;
    std::FILE* F(); // Produces checked pointer to f
public:
    File(); // Create object but don't open file
    File(const char* path, const char* mode = "r");
    ~File();
    int open(const char* path, const char* mode = "r");
    int reopen(const char* path, const char* mode);
    int getc();
    int ungetc(int c);
    int putc(int c);
    int puts(const char* s);
    char* gets(char* s, int n);
    int printf(const char* format, ...);
    size_t read(void* ptr, size_t size, size_t n);
    size_t write(const void* ptr, size_t size, size_t n);
    int eof();
    int close();
    int flush();
    int seek(long offset, int whence);
    int getpos(fpos_t* pos);
    int setpos(const fpos_t* pos);
    long tell();
    void rewind();
    void setbuf(char* buf);
    int setvbuf(char* buf, int type, size_t sz);
    int error();
    void clearErr();
};
#endif // FULLWRAP_H ///:~

```

这个类几乎包含了<stdio>中所有的文件I/O函数。(不包含**vfprintf()**，它只是用来实现**printf()**成员函数。)

类**File**的构造函数和前面的例子相同，并且这个类还有一个默认的构造函数。如果想创建**File**对象数组，或把**File**对象作为另一个类的数据成员来使用，这时类的初始化操作不在构造函数中完成，而是发生在其所属的对象已经创建之后，在这些情况下默认构造函数是很重要的。

默认构造函数将私有**FILE**指针**f**设为0。但是在对**f**进行任何引用之前，必须对其进行检查以确保指针不为空。这项操作由成员函数**F()**完成，这个函数为私有成员函数，这样做的目的是只允许类中的其他成员函数调用它。(不想让用户直接访问类的**FILE**结构。)

无论如何这并不是一种糟糕的解决方法。这种方法能起很好的作用，甚至能设想为标准(控制台)I/O和内核格式化(in-core formatting)(读/写一个内存块，而不是文件或控制台)构造相似的类。

在这里遇到的绊脚石是用于可变参数列表函数(variable argument list function)的运行时解释程序(runtime interpreter)。运行时解释程序是一段代码，它的作用是在运行时解析格式串(format string)，以及提取并解释从可变参数列表中得到的参数。产生这个问题有4个原因：

- 1) 即使仅仅需要使用解释程序的一小部分功能，该解释程序的所有内容也都会被加载到可执行程序中。所以，如果在程序中仅仅使用**printf("%c", 'x');**，那么程序包中所有的函数也都会被加载进来，包括打印浮点数和字符串的函数。没有标准选项可以减少程序使用的空间。

- 2) 因为解释是发生在运行时的，所以无法免除运行开销。这是很令人沮丧的，因为编译时所有的信息都存在格式串中，但是直到运行时刻才能对其进行求值。然而，如果能在编译时解析格式串中的变量，就可以产生直接的函数调用，速度比运行时解释程序更快(尽管**printf()**及同类函数已经很好地优化了)。

3) 因为格式串直到运行时才能求值, 所以可以没有编译时错误检查。如果读者曾经为找出函数 `printf()` 中的错误而对其使用错误的数字或者参数类型进行测试, 也许就对这个问题比较熟悉了。C++ 为尽早发现错误, 就进行编译时错误检查做了许多工作, 这使得代码的编写更加容易。把类型安全检查交给 I/O 库来完成似乎是欠妥的, 尤其是进行大量 I/O 操作时。

4) 对于 C++ 来说, 最关键的问题是 `printf()` 函数族不具备可扩展性。设计它们的目的是用来处理 C 语言中的基本数据类型 (`char`、`int`、`float`、`double`、`wchar_t`、`char*`、`wchar_t*` 和 `void*`) 以及这些数据类型的变体。读者也许会认为每次添加一个新类时, 可以重载函数 `printf()` 和 `scanf()` (以及它们的用于处理文件和字符串的变体), 但是请记住, 重载函数的参数列表中参数的类型必须不同, 然而 `printf()` 函数族把类型信息隐藏在可变参数列表和格式串中。对于一种语言如 C++ 来说, 如果设计它的目的是为了很容易地添加新的数据类型, 那么这个限制是无法接受的。

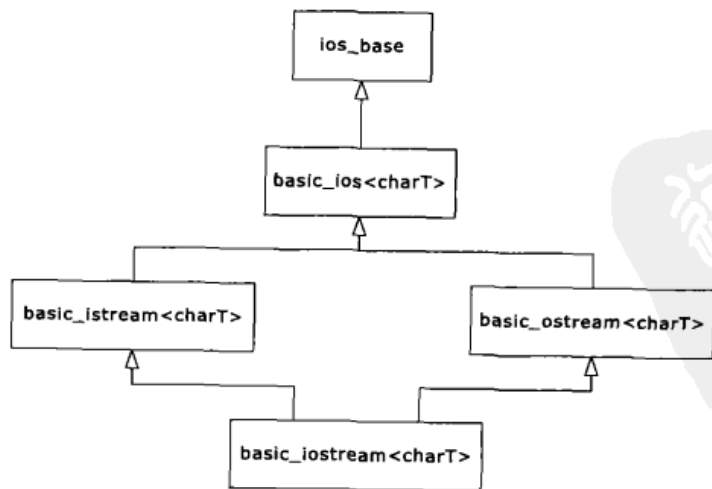
## 4.2 救助输入输出流

这些问题清楚地表明 I/O 是标准 C++ 类库最重要的内容之一。由于 “hello, world” 几乎是每个程序员学习一门新语言时所编写的第 1 个程序, 并且实际上每个程序都会用到 I/O, 所以 C++ 中的 I/O 类库必须非常易于使用。更大的挑战在于 I/O 类库必须适用于任何新的类。如此一来, 这个基础类库在设计时就需要一些技巧。除了能够学习到处理 I/O 和格式化操作用到的多种方法并提高其使用的准确性外之外, 在本章中读者还会看到一个真正功能强大的 C++ 类库是如何工作的。

### 4.2.1 插入符和提取符

流是一个传送和格式化固定宽度 (fixed width) 字符的对象。读者可以获得一个输入流 (通过 `istream` 类的子类)、一个输出流 (使用 `ostream` 对象) 或者同时实现两种功能的流 (使用从 `iostream` 派生的对象)。输入输出流类库提供了下面几种不同的类: 用于文件输入输出的 `ifstream`、`ofstream` 和 `fstream`, 用于标准 C++ 中 `string` 类输入输出的 `istringstream`、`ostreamstream` 和 `stringstream`。所有的这些流类拥有几乎相同的接口, 所以能够以统一的方式使用这些流类, 不管操作对象是文件、标准 I/O、内存区, 还是 `string` 对象。这样单一的接口同样支持扩充和增加一些新定义的类。某些函数实现格式化命令, 而某些函数以非格式化方式读写字符。

前面提到的流类实际上是模板的特化 (template specialization), 就像标准 `string` 类是 `basic_string` 模板的特化<sup>①</sup>。下图描述了输入输出流类继承体系中的基本类:



① 在第5章中深入讨论。

类**ios\_base**声明了所有流类共有的内容，不依赖于流所处理的字符类型。这些声明大部分是常量以及处理这些常量的函数，它们会在本章反复出现。其他类是以基础字符类型为参数的模板。例如类**istream**，定义如下：

```
typedef basic_istream<char> istream;
```

定义本章前面提及的类时，都使用了相似的类型定义（type definition）。另外，C++中也用**wchar\_t**（第3章中介绍的宽字符类型）来替换**char**定义了所有的输入输出流类。这在本章末尾可以看到。模板**basic\_ios**定义了输入和输出通用的函数，但是这依赖于基础字符类型（几乎不使用它们）。模板**basic\_istream**定义了一般的输入函数，**basic\_ostream**定义了一般的输出函数。后面介绍的文件流类和字符串流类增加了特殊的流处理功能。

在输入输出流类库中，重载了两种运算符以简化输入输出流的使用。运算符 **<<** 常用作输入输出流的插入符（inserter），运算符 **>>** 常用作提取符（extractor）。

提取符按照目标对象的类型解析输入信息。举例说明，可以使用**cin**对象，它是输入流，相当于C中的**stdin**，即可重定向标准输入（redirectable standard input）。在代码中包含头文件**<iostream>**时，就会预定义这个对象。

```
int i;
cin >> i;

float f;
cin >> f;

char c;
cin >> c;

char buf[100];
cin >> buf;
```

所有的内置数据类型都重载了**operator >>**。读者自己也可以重载**operator >>**，这将在后面看到。

为了显示不同变量中的内容，读者可以与插入符 **<<** 一起使用**cout**对象（相当于标准输出（standard output））；同样地，**cerr**对象相当于标准错误输出（standard error））：

```
cout << "i = ";
cout << i;
cout << "\n";
cout << "f = ";
cout << f;
cout << "\n";
cout << "c = ";
cout << c;
cout << "\n";
cout << "buf = ";
cout << buf;
cout << "\n";
```

尽管增强了类型检查功能，但是这样写出来的代码很乏味，而且似乎比用**printf( )**写出来的代码没有多大的提高。幸运的是，重载的插入符和提取符可以连续使用，构成复杂的表达式，使得写（和读）更容易：

```
cout << "i = " << i << endl;
cout << "f = " << f << endl;
cout << "c = " << c << endl;
cout << "buf = " << buf << endl;
```



为自己的类定义插入符和提取符，就是重载相关的运算符以完成正确的操作，即：

- 第1个参数定义成流（输入为**istream**，输出为**ostream**）的非**const**引用。
- 执行向/从流中插入/提取数据的操作（通过处理对象的组成元素）。
- 返回流的引用。

输入输出流应该是非常量，因为处理流数据将改变流的状态。通过返回流，如前所述，可以将这些流操作链接成单一的语句。

举个例子，考虑如何输出一个MM-DD-YYYY格式的**Date**类对象。下面的代码使用了插入符：

```
ostream& operator<<(ostream& os, const Date& d) {
    char fillc = os.fill('0');
    os << setw(2) << d.getMonth() << '-'
        << setw(2) << d.getDay() << '-'
        << setw(4) << setfill(fillc) << d.getYear();
    return os;
}
```

这个函数不能设为**Date**类的成员函数，因为运算符 `<<` 左边的操作数必须是输出流。**ostream**的成员函数**fill()**用于更换填充字符（padding character），当输出域（field）的宽度大于输出数据长度时，使用填充字符填充超出部分，域宽由操纵算子（manipulator）**setw()**决定。使用“0”作为前导填充字符，所以显示10月之前的月份时，如显示9月份为“09”。函数**fill()**返回原有的填充字符（默认为一个空格符），以便在后面使用操纵算子**setfill()**恢复这个填充字符。本章后面将深入讨论操纵算子。

使用提取符需要注意输入数据错误。通过设置流的失败标志位（fail bit）可以表明产生了流错误，如下所示：

```
istream& operator>>(istream& is, Date& d) {
    is >> d.month;
    char dash;
    is >> dash;
    if(dash != '-')
        is.setstate(ios::failbit);
    is >> d.day;
    is >> dash;
    if(dash != '-')
        is.setstate(ios::failbit);
    is >> d.year;
    return is;
}
```

一旦流的失败标志位被设置，则在流恢复到有效状态之前，此外所有的流操作都会被忽略（简要说明一下）。这就是为什么即使设置了**ios::failbit**，上述代码也继续进行提取操作。这种实现有些宽松，因为它允许在日期字符串的数字和短线之间插入空格（因为在默认情况下 `>>` 运算符在读取内置数据类型时会跳过空格）。对提取符来说，下面是合法的日期字符串：

```
"08-10-2003"
"8-10-2003"
"08 - 10 - 2003"
```

下面是非法的日期字符串：

```
"A-10-2003" // No alpha characters allowed
"08%10/2003" // Only dashes allowed as a delimiter
```

将会在4.3节处理流错误部分深入讨论流状态。



### 4.2.2 通常用法

正如**Date**提取符所展示的，必须防止错误的输入。如果输入一个非法的值，处理过程就会出现错误，并且很难恢复。另外，默认情况下的格式化输入使用空格作为界定符。把前面的代码片断合成一个单独的程序，看看会发生什么情况：

```
//: C04:Iosexamp.cpp {RunByHand}
// Iostream examples.
#include <iostream>
using namespace std;

int main() {
    int i;
    cin >> i;

    float f;
    cin >> f;

    char c;
    cin >> c;

    char buf[100];
    cin >> buf;

    cout << "i = " << i << endl;
    cout << "f = " << f << endl;
    cout << "c = " << c << endl;
    cout << "buf = " << buf << endl;

    cout << flush;
    cout << hex << "0x" << i << endl;
} ///:~
```

给出如下输入：

```
12 1.4 c this is a test
```

预期的输出：

```
12
1.4
c
this is a test
```

但是输出和预期的有些不同：

```
i = 12
f = 1.4
c = c
buf = this
0xc
```

注意，**buf**仅得到了第1个单词，因为输入机制是通过寻找空格来分割输入的，而空格出现在 **"this"** 的后面。另外，如果连续输入的字符串长度超过**buf**的存储空间，就会发生**buf**溢出现象。

实际上在交互程序中，经常需要一次输入一行字符序列，当这些字符安全地存储到缓冲区后再进行扫描和转换工作。使用这种方法，读者不必担心输入程序的执行因非法数据的出现而阻塞。

另一个需要考虑的内容是在命令行界面的输入输出。这仅在过去控制台比一台打字机强不

了多少的时候才有意义，但是现在图形用户界面（graphical user interface, GUI）迅速占据了统治地位。在这样的环境下使用控制台I/O是否还有意义呢？除了很简单的例子或测试外，可以完全忽略`cin`并采用下面的方法：

1) 如果程序需要输入，则从文件中读入数据——读者很快就会看到通过输入输出流来使用文件非常容易。文件输入输出流在图形用户界面下也能很好地工作。

2) 正如我们建议的那样，读取输入但不试图对其进行转换。当输入数据被保存到某处，并且在转换时不会造成错误时，才可以安全地扫描它。

3) 输出的情况有所不同。如果使用图形用户界面，就不需要用`cout`，必须把数据输出到文件（和输出到`cout`是一样的），或者使用图形用户界面应用程序实现数据显示。否则，把数据输出到`cout`便很有意义。在这两种情况下，输入输出流的输出格式化功能十分有用。

在大型项目中，另一个常用的方法可以节省编译时间。例如，看看本章前面是如何在头文件中声明**Date**流操作符的。仅需要包含函数的原型，不需要在**Date.h**中包含整个**<iostream>**头文件。标准的方法是像下面这样仅声明类：

```
class ostream;
```

这是一种将接口从实现中分离的早就在频繁使用的技巧，称作前置声明（forward declaration），在其出现的位置上，**ostream**应当被视为未完成的类型，因为这时编译器还没有看到类的定义）。

然而，这样的声明不能正常工作，有两个原因：

- 1) 流类是在名字空间**std**中定义的。
- 2) 这些流类是模板。

正确的声明应该是：

```
namespace std {
    template<class charT, class traits = char_traits<charT> >
        class basic_ostream;
    typedef basic_ostream<char> ostream;
}
```

（正如读者所看到的，就像**string**类，流类使用了第3章中提到的字符特征类。）由于为所有需要引用的流类编写代码是十分枯燥乏味的，C++标准提供了头文件**<iosfwd>**来完成这些工作。**Date**头文件如下所示：

```
// Date.h
#include <iosfwd>

class Date {
    friend std::ostream& operator<< (std::ostream&,
                                    const Date&);
    friend std::istream& operator>> (std::istream&, Date&);
    // Etc.
```

#### 4.2.3 按行输入

有3种可选的方法来实现按行输入：

- 成员函数**get()**
- 成员函数**getline()**
- 定义在头文件**<string>**中的全局函数**getline()**

前两个函数有3个参数：



- 1) 指向字符缓冲区的指针，用于保存结果。
- 2) 缓冲区的大小（为了保证缓冲区不会溢出）。
- 3) 结束字符，根据结束字符判断何时停止读入操作。

结束字符（terminating character）默认情况下为'\n'，这是常用的结束字符。当在输入过程中遇到结束字符时，这两个函数都会在结果缓冲区末尾存储一个零。

那么，**get()**和**getline()**两个函数有什么不同呢？细微而重要的区别在于：当遇到输入流中的界定符（delimiter，即结束字符）时**get()**停止执行，但是并不从输入流中提取界定符。这时，如果再次调用**get()**会遇到同一个界定符，函数将立即返回而不会提取输入。（为了解决这个问题，据推测，需要在下一个**get()**函数中使用不同的界定符或使用不同的输入函数。）函数**getline()**则相反，它将从输入流中提取界定符，但仍然不会把它存储到结果缓冲区中。

<string>中定义的函数**getline()**使用起来很方便。它不是成员函数，而是在名字空间**std**中声明的独立函数。这个函数仅有两个非默认参数，输入流和**string**对象。从函数名可以看出，它从输入流中读取字符直到遇到第1个界定符（默认为'\n'）并且丢弃这个界定符。这个函数的优点在于它把数据读入一个**string**对象中，所以不用担心缓冲区的大小。

一般来说，当采用按行输入的方式处理文本文件时，需要使用其中的一个**getline()**函数。

#### 1. **get()**函数的重载版本

函数**get()**也有另外3个重载版本：其中一个版本没有参数，使用**int**作为返回值类型，返回下一个字符；另一个版本使用**char**类型的引用作为参数，函数从流中读取一个字符放到这个参数中；还有一个版本则把流类对象直接存储到基础的缓冲区结构。本章后面将对最后一个版本进行深入研究。

#### 2. 读原始字节

如果准确知道正在处理的数据并需要把字节直接移动到内存中一个变量、数组或结构中，可以使用非格式化的I/O函数**read()**。这个函数的第1个参数是指向目标内存的指针，第2个参数是需要读入的字节数。如果预先把信息保存在文件中，例如使用输出流的**write()**成员函数将信息保存为二进制形式（当然，使用相同的编译器），那么这个函数就十分有用。在后面读者会看到这些函数的例子。

### 4.3 处理流错误

前面涉及的**Date**提取符在某种情况下将设置流的失败位。那么，用户如何知道这样的失败是何时发生的呢？可以通过调用流的成员函数来测试流错误，或者如果不关心到底发生了什么错误，你可以只是在一个布尔环境中评估该流。这两种技术都依赖于流的错误位的状态。

#### 1. 流状态

类**ios\_base**从类**ios**派生而来<sup>①</sup>，定义了4个标志位来测试流的状态：

标志位	意 义
<b>badbit</b>	发生了（或许是物理上的）致命性错误。流将不能继续使用 输入结束（文件流的物理结束或用户结束了控制台流输入，例如用户按下了Ctrl-Z 或 Ctrl-D）
<b>eofbit</b>	
<b>failbit</b>	I/O操作失败，主要原因是非法数据（例如，试图读取数字时遇到字母）。流可以继续使用。输入结束时也将设置 <b>failbit</b> 标志
<b>goodbit</b>	一切正常；没有错误发生。也没有发生输入结束

① 由于这个原因，可以使用**ios::failbit** 取代**ios\_base::failbit**以节省输入。

可以通过调用相应成员函数，根据其返回的布尔值来测试发生了什么情况，返回的布尔值指出设置了哪个标志位。当除了**goodbit**之外的其他3个标志位没有被设置时，流类的成员函数**good()**返回真(**true**)。如果**eofbit**被设置则函数**eof()**返回真，表明程序试图从已经到达末尾的流(通常是文件流)中读取数据。因为输入结束(end-of-input)发生在C++的读操作试图越过物理介质末尾的时候，所以**failbit**标志位也会被设置，表示期望获取的数据没有被成功读取。如果设置了**failbit**或**badbit**标志位中的任意一个，则函数**fail()**返回真，只有设置了**badbit**标志位，函数**bad()**才返回真。

一旦设置了流状态中的任何一个标志位，这些标志位将保持不变，但程序员并不希望总保持这种状况。读文件时，也许想在文件结束之前将文件指针重定位到前面的位置。仅靠移动文件指针不会自动重置**eofbit**或**failbit**标志位。需要程序员自己在程序中调用**clear()**函数来清空标志位，如下所示：

```
myStream.clear(); // Clears all error bits
```

调用**clear()**后，立即调用函数**good()**则返回**true**。正如较早时提及的**Date**提取符一样，函数**setstate()**用来设置标志位，而标志位的值由我们传递给它。函数**setstate()**不会影响其他标志位，如果已经设置了其他的标志位，则这些标志位将保持不变。如果想设置某个特定的标志位而重置其他所有的标志位，可以调用重载的**clear()**函数，将需要设置的标志位的表达式作为参数传递给它，如下所示：

```
myStream.clear(ios::failbit | ios::eofbit);
```

在大多数情况下，程序员不想逐个检查流状态位，只想知道所有操作是否成功完成。当从头到尾读文件时就是这种情况，此时只想知道什么时候读文件完成。可以使用返回值为**void\***的转换函数，当流出现在布尔表达式中时，这个函数被自动调用。使用下面的语句完成流的读入，直到输入结束：

```
int i;
while(myStream >> i)
    cout << i << endl;
```

记住，**operator>>()**返回它的流参数，所以上面的**while**语句用布尔表达式对流进行测试。这个特殊的例子假定输入流**myStream**包含由空格符分隔的整数。函数**ios\_base::operator void\*()**仅在流上调用函数**good()**并返回调用结果<sup>①</sup>。因为大部分流操作返回流对象，所以使用这个语句是比较方便的。

## 2. 流类和异常

在出现异常概念之前的很长时间内，输入输出流是作为C++的一部分存在的，所以一直采用手工方式检查流状态。为了保持向后兼容性，这种手工检查流状态的方法仍能在程序中使用，但是现代的输入输出流采用抛出异常的方法来代替它。流的成员函数**exceptions()**接受一个参数，这个参数用于表示程序员希望在哪个状态位出现时抛出异常。当遇到这样的状态时，就抛出一个**std::ios\_base::failure**类型的异常，**std::ios\_base::failure**继承自**std::exception**。

尽管可以为4种流状态中的任何一个状态触发失败异常，但是这样做并不是好办法。如第1章所述，要在真正发生异常的情况下使用异常，但是“已至文件尾”不仅不是异常，而且是在

① 习惯上优先使用函数**operator void\*()**而不是**operator bool()**，因为从**bool**型隐式转换到**int**型会引起错误，在用整形表达式时，不应该错误地应用流。函数**operator void\*()**在布尔表达式中应该隐式调用。

预料之中的正常情况。因此读者也许只想对**badbit**所代表的错误使用异常，使用方法如下：

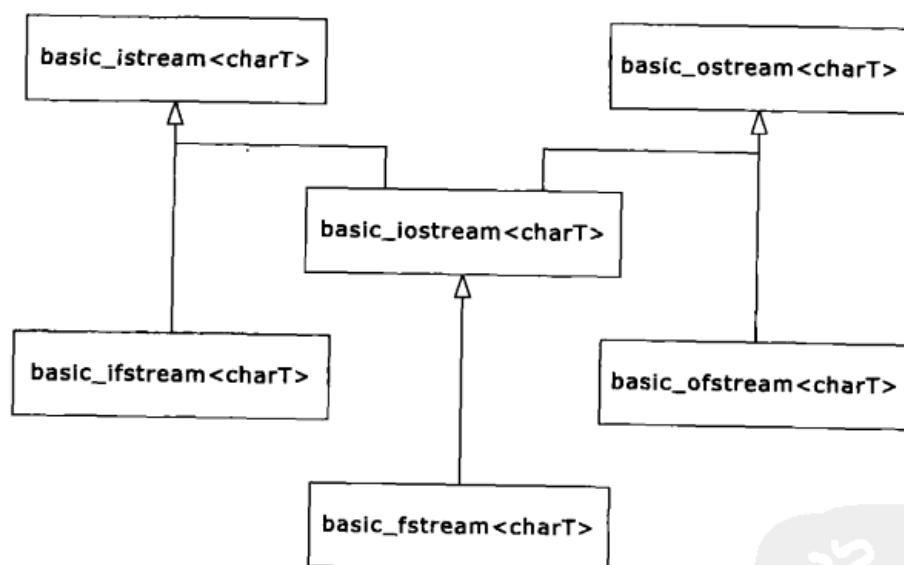
```
myStream.exceptions(ios::badbit);
```

在流接流（stream-by-stream）的基础上能使用异常，因为**exceptions()**是流类的成员函数。函数**exceptions()**返回位掩码（bitmask）<sup>①</sup>（**iostate**类型，它依赖于编译器，可转成**int**型），表示哪种流状态会触发异常。如果这些状态位已经设置，就会立即抛出异常。当然，如果与流一起使用异常，则最好捕获异常，这意味着需要把流处理过程封装到含有**ios::failure**处理器的**try**块中。许多程序员认为这很乏味，他们只在发生错误的地方手工检查错误状态（因此假如，大部分情况下他们不希望**bad()**返回**true**）。这是让流抛出异常成为可选项而不是默认项的另一原因。在任何情况下都可以选择处理流错误的方式。基于相同的原因，我们推荐使用异常进行错误处理，这里也采用这种方法。

## 4.4 文件输入输出流

使用输入输出流类操纵文件比使用C语言中的**stdio**更容易、更安全。打开一个文件要做的全部工作就是创建一个对象——这是构造函数所做的工作。不需要显式地关闭文件（尽管能使用成员函数**close()**来关闭文件），因为当对象超出作用域时析构函数会关闭文件。构造一个**ifstream**对象用于创建默认的输入文件。构造一个**ofstream**对象用于创建默认的输出文件。一个**fstream**对象既可以用于输入文件，也可以用于输出文件。

下图说明了适用于输入输出流类的文件流类：



和以前一样，这里实际使用的类都是由类型定义的模板的特化。例如，**ifstream**用来处理**char**文件，定义如下：

```
typedef basic_ifstream<char> ifstream;
```

### 4.4.1 一个文件处理的例子

下面这个例子展示了迄今为止所讨论到的所有特性。注意，对**<fstream>**的包含声明了文件输入输出类。尽管在许多平台上，这样的声明将自动包含**<iostream>**，但编译器不一定

<sup>①</sup> 用作单个标志位的一个整数类型。

都这样做。如果想写出可移植的代码，则这两个头文件都要包含进来：

```
//: C04:Strfile.cpp
// Stream I/O with files;
// The difference between get() & getline().
#include <fstream>
#include <iostream>
#include "../require.h"
using namespace std;

int main() {
    const int SZ = 100; // Buffer size;
    char buf[SZ];
    {
        ifstream in("Strfile.cpp"); // Read
        assure(in, "Strfile.cpp"); // Verify open
        ofstream out("Strfile.out"); // Write
        assure(out, "Strfile.out");
        int i = 1; // Line counter

        // A less-convenient approach for line input:
        while(in.get(buf, SZ)) { // Leaves \n in input
            in.get(); // Throw away next character (\n)
            cout << buf << endl; // Must add \n
            // File output just like standard I/O:
            out << i++ << ": " << buf << endl;
        }
    } // Destructors close in & out

    ifstream in("Strfile.out");
    assure(in, "Strfile.out");
    // More convenient line input:
    while(in.getline(buf, SZ)) { // Removes \n
        char* cp = buf;
        while(*cp != ':')
            ++cp;
        cp += 2; // Past ": "
        cout << cp << endl; // Must still add \n
    }
} ///:~
```

创建**ifstream**对象和**ofstream**对象后都调用了函数**assure()**，以确保文件被成功打开。在编译器希望产生结果为布尔值的地方，流对象产生了表示成功或失败的值。

第1个**while**循环演示了两个**get()**函数的使用。第1个**get()**读字符到缓冲区，当已经读取了**SZ-1**个字符或者读取字符过程中遇到了第3个参数（默认为'\n'）所规定的字符时，在缓冲区中写入零结束符。函数**get()**跳过输入流中的结束字符，所以需使用没有参数的**get()**，通过调用**in.get()**丢掉结束字符，这个函数读取一个字节，并返回一个**int**型的值。也可以使用具有两个默认参数的成员函数**ignore()**。它的第1个参数表示要跳过的字符的数目，默认值为1。第2个参数指明遇到哪个字符时，函数**ignore()**退出（在提取这个字符之后），默认值为**EOF**。

紧接着是两个相似的输出语句：一个输出到**cout**，一个输出到文件**out**。注意，使用这个方法很方便，不需要担心对象类型，因为格式化语句对所有**ostream**对象都能很好地处理。前一条语句把一行显示到标准输出上，第2条语句将一行以及其行号写入一个新文件中。

为演示函数**getline()**如何工作，现在打开刚刚创建的文件，跳过行号。在以读方式再次打开文件之前，为了确保正确地关闭这个文件，这里有两种方法可供选择。可以使用花括号把程序的第1部分括起来，从而强制对象**out**超出作用域，这会使系统调用析构函数并关闭该文

件，这个程序就是这样做的。也可以调用成员函数**close()**关闭这些文件；如果采用这种方法，甚至可以通过调用成员函数**open()**重用对象**in**。

第2个**while**循环语句说明了函数**getline()**遇到输入流中的结束字符（函数的第3个参数，默认为'\n'）时是如何将其除去的。尽管**getline()**像**get()**一样在缓冲区中写入字符零，但它不在缓冲区插入结束字符。

这个例子，连同本章的大部分例子，都假定对任何重载函数**getline()**的调用都会遇到新行界定字符。如果不是这种情况，流的状态位**eofbit**就会被设置，并且对**getline()**的调用返回**false**，造成程序丢失该输入的最后一行字符。

#### 4.4.2 打开模式

通过覆盖构造函数的默认参数，可以控制文件的打开模式。下表说明了控制文件打开模式的标志：

标 志	功 能
<b>ios::in</b>	打开输入文件。将这种打开模式应用于 <b>ofstream</b> 可以使现存文件不被截断
<b>ios::out</b>	打开输出文件。将这种模式应用于 <b>ofstream</b> ，而且没有使用 <b>ios::app</b> 、 <b>ios::ate</b> 或 <b>ios::in</b> 时，意味着使用的是 <b>ios::trunc</b> 模式
<b>ios::app</b>	打开一个仅用于追加的输出文件
<b>ios::ate</b>	打开一个已存在文件（输入或输出），并把文件指针指向文件末尾
<b>ios::trunc</b>	如果文件存在，则截断旧文件
<b>ios::binary</b>	以二进制方式打开文件。默认打开为文本方式

位掩码（bitwise）或运算符可以使用这些标记的组合。

二进制标记只在一些非UNIX系统上起作用，例如从MS-DOS发展而来的操作系统，这些系统对行结束界定符有特殊约定。例如，在MS-DOS系统的文本模式（默认模式）下，每输出一个换行符('\n')，文件系统实际上输出了两个字符，一个回车符/换行符（CRLF）对，ASCII码为0x0D和0x0A。相反，当在文本模式下读这样的文件到内存，遇到这样的字节对时，就在相应的位置写入一个'\n'来替代。如果想绕过这个特殊的处理过程，可以采用二进制模式打开文件。在二进制模式下，不论是否写生僻的字节到文件，都没有需要特殊处理的事情，总是能进行操作（通过调用**write()**）。然而，应该在使用**read()**或**write()**的时候以二进制模式打开文件，因为这些函数有一个每次读/写字节个数的参数。在这些情况下，如果流中包含额外字符'\r'，字节个数参数将不再起作用。如果想使用本章后面将要讨论的流指针定位命令，则也应该使用二进制模式打开文件。

可以通过声明**fstream**对象打开一个文件，同时用作输入和输出。声明**fstream**对象的时候，必须使用足够的打开模式标记，告诉文件系统现在想进行输入、输出或既输入又输出。从输出切换到输入的时候，需要刷新流或者重定位文件指针。从输入切换到输出，也需要重定位文件指针。通过**fstream**对象创建一个文件，在构造函数中使用**ios::trunc**打开模式标志，可以调用输入和输出文件。

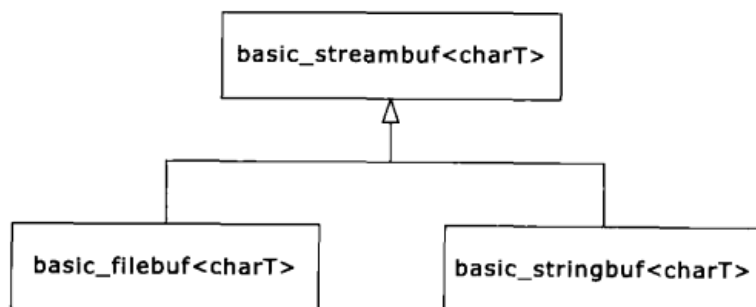
#### 4.5 输入输出流缓冲

好的设计原则指出，无论何时创建一个新类，都应该尽最大努力隐藏实现细节。只让用户看到他们需要知道的东西，将其他东西都设为**private**以避免引起混乱。使用插入符和提取符

时，一般程序员不知道或不关心数据在哪里产生和消亡，不管处理的对象是标准I/O、文件、内存还是新创建的类或设备。

然而，重要的是与产生和消耗数据的输入输出流部分进行通信。为给这部分提供统一的接口并隐藏底层实现，标准库把它抽象成一个类，称为**streambuf**。每个输入输出流对象包含一个指向**streambuf**的指针。（对象类型依赖于被处理的内容是标准I/O、文件还是内存等。）用户可以直接访问**streambuf**；例如，可以把原始字节移入或移出**streambuf**，而不用通过封装它的输入输出流对其进行格式化。这可以通过调用**streambuf**对象的成员函数来完成。

目前，读者需要知道的最重要的事情，就是每个输出流对象包含一个指向**streambuf**对象的指针，并且**streambuf**对象拥有一些可供调用的成员函数。对于文件流类和字符串流类，他们有特殊的流缓冲区类型，如下图所示：



为了能够访问**streambuf**，每个输入输出流对象有一个成员函数**rdbuf()**，它返回一个指向对象**streambuf**的指针，通过这个指针可以对**streambuf**对象进行存取。这样就可以对基础的**streambuf**调用任何成员函数。然而，更有趣的是，可以把**streambuf**指针和另一个输入输出流对象用运算符 `<<` 连接到一起。这将把右侧对象中的字符都输出到运算符 `<<` 左侧的对象中去。这样一来，如果想把一个输入输出流中的字符全部移动到另一个输入输出流中，就不需要通过令人厌烦的方法一次读一个字符或一行字符（会引起潜在的错误）。这是一种很优雅的方法。

下面是一个简单的示例程序，该程序打开一个文件并把文件中的内容送到标准输出（和前面的例子相似）：

```

//: C04:Stype.cpp
// Type a file to standard output.
#include <fstream>
#include <iostream>
#include "../require.h"
using namespace std;

int main() {
    ifstream in("Stype.cpp");
    assure(in, "Stype.cpp");
    cout << in.rdbuf(); // Outputs entire file
} ///:~

```

在这里使用这个程序的源代码文件作为参数创建了一个**ifstream**对象。如果文件不能打开，则函数**assure()**报告打开文件失败。所有的工作实际上都发生在如下语句中：

```
cout << in.rdbuf();
```

这条语句把文件中的全部内容输出到**cout**。这样的语句不仅使得代码更简洁，而且常常比一



次移动若干字节效率更高。

一种形式的**get()**函数直接把数据写入另一个对象的**streambuf**中。这个函数的第1个参数是目标**streambuf**的引用，第2个参数是使函数**get()**停止执行的结束字符（默认情况下为‘\n’）。如下所示，还有另外一种将文件发送到标准输出的方法：

```
//: C04:Sbufget.cpp
// Copies a file to standard output.
#include <fstream>
#include <iostream>
#include "../require.h"
using namespace std;

int main() {
    ifstream in("Sbufget.cpp");
    assure(in);
    streambuf& sb = *cout.rdbuf();
    while(!in.get(sb).eof()) {
        if(in.fail())           // Found blank line
            in.clear();
        cout << char(in.get()); // Process '\n'
    }
} ///:~
```

函数**rdbuf()**返回一个指针，它必须解除引用（dereference），以满足函数查看对象的需要。流缓冲区不会被复制（它们没有拷贝构造函数），所以将**sb**定义为**cout**的流缓冲区的引用。并调用函数**fail()**和**clear()**以处理输入文件中的空行（在这个例子中就是如此）。当这个特殊的重载函数**get()**看到在一行中有两个新界定符（一个空行的证据），就设置输入流的失败位，所以必须调用**clear()**来重置失败位以继续从流中读取数据。对**get()**的第2次调用提取并回应每个新行的界定字符。（记住，**get()**和**getline()**提取界定字符的方法不同。）

也许并不需要经常使用这种技术，但是知道这种方法总是有益处的<sup>①</sup>。

## 4.6 在输入输出流中定位

每种类型的输入输出流都有“下一个”字符从哪里来（如果是**istream**）或到哪里去（如果是**ostream**）的概念。在某些情况下，需要移动流的位置（通过设置“流指针”给流重新定位）。可以使用两种方式进行流定位：一种是使用称为**streampos**的流指针进行绝对流位置定位；另一种方式和标准C中用于处理文件的库函数**fseek()**相似，实现从文件头、文件尾或当前位置移动某个给定的字节数进行相对流定位。

用**streampos**进行绝对流位置定位，需要先调用一个“告知”函数，以便知道流指针在流中的确切位置：对于**ostream**调用**tellp()**，对于**istream**调用**tellg()**。（“p”表示“写指针”，“g”表示“读指针”。）这个函数返回一个**streampos**，稍后当要回到流中定位流指针时要用到它，对**ostream**对象调用**seekp()**，对**istream**对象调用**seekg()**。

第2种方法是相对定位，使用重载版本的**seekp()**和**seekg()**函数。函数的第1个参数是要移动的字符数目：这个数目可正可负。第2个参数是移动方向：

① 关于流缓冲区和流的更深入的讨论可参考Addison-Wesley出版社1999年出版的Langer & Kreft的《Standard C++ Iostreams and Locales》。

<b>ios::beg</b>	流的开始位置
<b>ios::cur</b>	流的当前位置
<b>ios::end</b>	流的末端位置

下面是在文件中进行定位的一个例子，但是这里没有C语言中**stdio**对于在文件中进行定位所做的那些限制。在C++中，可以在任何类型的输入输出流中进行定位（尽管在标准流对象如**cin**和**cout**中不允许这样做）：

```
//: C04:Seeking.cpp
// Seeking in iostreams.
#include <cassert>
#include <cstdint>
#include <cstring>
#include <fstream>
#include "../require.h"
using namespace std;

int main() {
    const int STR_NUM = 5, STR_LEN = 30;
    char origData[STR_NUM][STR_LEN] = {
        "Hickory dickory dus. . .",
        "Are you tired of C++?",
        "Well, if you have,",
        "That's just too bad,",
        "There's plenty more for us!"
    };
    char readData[STR_NUM][STR_LEN] = {{ 0 }};
    ofstream out("Poem.bin", ios::out | ios::binary);
    assure(out, "Poem.bin");
    for(int i = 0; i < STR_NUM; i++)
        out.write(origData[i], STR_LEN);
    out.close();
    ifstream in("Poem.bin", ios::in | ios::binary);
    assure(in, "Poem.bin");
    in.read(readData[0], STR_LEN);
    assert(strcmp(readData[0], "Hickory dickory dus. . .")
        == 0);
    // Seek -STR_LEN bytes from the end of file
    in.seekg(-STR_LEN, ios::end);
    in.read(readData[1], STR_LEN);
    assert(strcmp(readData[1], "There's plenty more for us!")
        == 0);
    // Absolute seek (like using operator[] with a file)
    in.seekg(3 * STR_LEN);
    in.read(readData[2], STR_LEN);
    assert(strcmp(readData[2], "That's just too bad,") == 0);
    // Seek backwards from current position
    in.seekg(-STR_LEN * 2, ios::cur);
    in.read(readData[3], STR_LEN);
    assert(strcmp(readData[3], "Well, if you have,") == 0);
    // Seek from the beginning of the file
    in.seekg(1 * STR_LEN, ios::beg);
    in.read(readData[4], STR_LEN);
    assert(strcmp(readData[4], "Are you tired of C++?")
        == 0);
} ///:-
```

这个程序使用二进制输出流写一首诗到文件中。重新打开**ifstream**文件后，使用**seekg()**“获取指针”位置。正如读者所看到的，文件指针可以从文件头、文件尾或从文件当前位置进

行搜索。显然，从文件头开始移动指针需要提供一个正数做参数，而从文件尾移动指针需要提供一个负数做参数。

既然已经熟悉了**streambuf**类以及如何在流中定位，读者就能理解创建一个既能够读文件又能够写文件的流对象的另一种方法了（不使用**fstream**对象）。下面的代码首先使用某些标记创建一个**ifstream**，这些标记表明它既能用于文件输入又能用于文件输出。因为不能对**ifstream**对象进行写入操作，所以需要创建一个具有内置流缓冲区的**ostream**对象：

```
ifstream in("filename", ios::in | ios::out);
ostream out(in.rdbuf());
```

读者也许想知道向这两个对象之一写入数据时会发生什么情况。举例如下：

```
//: C04:Iofile.cpp
// Reading & writing one file.
#include <fstream>
#include <iostream>
#include "../require.h"
using namespace std;

int main() {
    ifstream in("Iofile.cpp");
    assure(in, "Iofile.cpp");
    ofstream out("Iofile.out");
    assure(out, "Iofile.out");
    out << in.rdbuf(); // Copy file
    in.close();
    out.close();
    // Open for reading and writing:
    ifstream in2("Iofile.out", ios::in | ios::out);
    assure(in2, "Iofile.out");
    ostream out2(in2.rdbuf());
    cout << in2.rdbuf(); // Print whole file
    out2 << "Where does this end up?";
    out2.seekp(0, ios::beg);
    out2 << "And what about this?";
    in2.seekg(0, ios::beg);
    cout << in2.rdbuf();
} ///:~
```

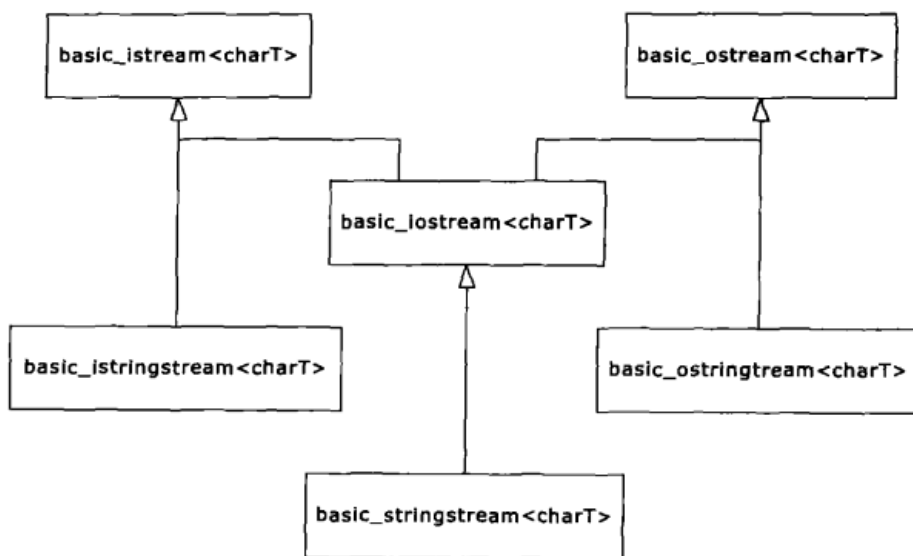
前5行把这个程序的源代码拷贝到一个叫做**iofile.out**的文件，接着关闭该文件，这样一来，就为读者提供一个安全的文本文件。然后，使用前面介绍的技术创建两个对象，以便对同一个文件进行读和写操作。在语句**cout<<in2.rdbuf()**中，可以看到“读”指针被初始化为指向文件头。“写”指针被设置为指向文件尾，因为文本信息“Where does this end up?”是追加到文件中的。然而，如果写指针通过调用函数**seekp()**被移动到指向文件头，新写入的文本将覆盖原来的文本。调用函数**seekg()**把读指针移回到文件头，就可以分别看到两次写操作后所显示的文件中的内容。当**out2**超出作用域范围后，系统调用析构函数，使文件自动保存和关闭。

## 4.7 字符串输入输出流

字符串输入输出流类直接对内存而不是对文件和标准输出（设备）进行操作。它使用与**cin**及**cout**相同的读取和格式化函数来操纵内存中的数据。在早期的计算机中，内存存储器是计算机的核心，所以这种功能的类型常常称为内核格式化（in-core formatting）。

字符串流的类名模仿文件流的类名。如果需要创建一个字符串流，并从这个流读取字符，可以创建**istream**。如果需要写字符到字符串流，可以创建**ostream**。所有字

字符串流类的声明都包含在标准头文件<sstream>中。照例，有一些类模板被添加到输入输出流类层次结构中，如下图所示：



#### 4.7.1 输入字符串流

为了使用流操作从一个字符串中读取数据，需要创建**istringstream**对象并用字符串初始化这个对象。下面的程序说明了如何使用**istringstream**对象：

```

//: C04:Istring.cpp
// Input string streams.
#include <cassert>
#include <cmath> // For fabs()
#include <iostream>
#include <limits> // For epsilon()
#include <sstream>
#include <string>
using namespace std;

int main() {
    istringstream s("47 1.414 This is a test");
    int i;
    double f;
    s >> i >> f; // Whitespace-delimited input
    assert(i == 47);
    double relerr = (fabs(f) - 1.414) / 1.414;
    assert(relerr <= numeric_limits<double>::epsilon());
    string buf2;
    s >> buf2;
    assert(buf2 == "This");
    cout << s.rdbuf(); // " is a test"
} ///:~

```

读者可以看到，这是一种将字符串转换为特定类型值的方法，比标准C中的库函数如**atof()**和**atoi()**更灵活、更通用，尽管后者对单个字符串的转换效率更高。

在表达式**s >> i >> f**中，字符串中的第1个数字摘录到**i**，第2个数字输出到**f**。因为它依赖于数据的类型进行摘录，所以不是“以空格作为首选界定符的字符集”。例如，当字符串为“**1.414 47 This is a test**”时，则**i**提取的值为**1**，因为输入过程会在小数点处停止。**f**提取的值为**0.414**。如果想把一个浮点数分成整数部分和小数部分，则这种方法就很有效。而在其他

情况下这似乎是错误的做法。第2个**assert()**函数判断读出的数据是否发生了错误；这样做比简单地比较两个浮点数是否相等更好。函数**epsilon()**返回一个在<limits>中定义的常量，代表双精度数字的机器误差，这是在比较两个**double**型数据时所能得到的最小误差<sup>①</sup>。

或许读者已经猜到**buf2**中的内容不等于剩下的串，只是等于下一个空格字符之前的单词。一般来说，当知道输入流中数据的顺序并把它转换成除字符串之外的数据类型时，最好使用输入输出流提取符。然而，如果想一次性提取一个串中剩余的部分并把它输出到另一个输入输出流中，可以使用程序中所示的函数**rdbuf()**。

为测试本章早些时候提到的**Date**提取符，在下面的测试程序中使用输入字符串流：

```
//: C04:DateIOTest.cpp
//{L} ../C02/Date
#include <iostream>
#include <sstream>
#include "../C02/Date.h"
using namespace std;

void testDate(const string& s) {
    istringstream os(s);
    Date d;
    os >> d;
    if(os)
        cout << d << endl;
    else
        cout << "input error with \"" << s << "\"" << endl;
}

int main() {
    testDate("08-10-2003");
    testDate("8-10-2003");
    testDate("08 - 10 - 2003");
    testDate("A-10-2003");
    testDate("08%10/2003");
} ///:~
```

在**main()**函数中，将每个字符串作为引用参数依次调用函数**testDate()**，函数**testDate()**把参数封装到**istringstream**对象中，这样就可以测试为**Date**对象所编写的流提取符了。函数**testDate()**也对插入符**operator<<()**进行了测试。

#### 4.7.2 输出字符串流

为了创建输出字符串流，可以构造**ostringstream**对象，这个类对象可以管理动态字符缓冲区，缓冲区存放要插入的任何字符串。为了将输出结果格式化为**string**对象，可以调用成员函数**str()**，举例如下：

```
//: C04:Ostring.cpp {RunByHand}
// Illustrates ostringstream.
#include <iostream>
#include <sstream>
#include <string>
using namespace std;
int main() {
    cout << "type an int, a float and a string: ";
```

① 关于机器双精度数字和浮点数的一般性计算，请参考“标准C库，第三部分”，C/C++用户周刊，1995年3月。也可查阅网页：[www.freshsources.com/1995006a.html](http://www.freshsources.com/1995006a.html)。

```

int i;
float f;
cin >> i >> f;
cin >> ws; // Throw away white space
string stuff;
getline(cin, stuff); // Get rest of the line
ostringstream os;
os << "integer = " << i << endl;
os << "float = " << f << endl;
os << "string = " << stuff << endl;
string result = os.str();
cout << result << endl;
} ///:~

```

这个例子中读取**int**和**float**的语句与**Istring.cpp**中的语句相似。下面是一个输出结果的例子（黑体部分为键盘输入）。

```

type an int, a float and a string: 10 20.5 the end
integer = 10
float = 20.5
string = the end

```

读者可以看到，就像其他的输出流，输出字节到**ostringstream**对象可以使用一般的格式化工具，如运算符**<<**和**endl**。每次调用**str()**函数都会返回一个新的**string**对象，所以字符串流内置的**stringbuf**对象不会被破坏。

在第3章中，给出了一个程序**HTMLStripper.cpp**，它的作用是删除文本文件中的所有HTML标记及特殊字符。这里给出一种使用字符串流的更优美的版本：

```

//: C04:HTMLStripper2.cpp {RunByHand}
//{L} ../C03/ReplaceAll
// Filter to remove html tags and markers.
#include <cstdint>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <sstream>
#include <stdexcept>
#include <string>
#include "../C03/ReplaceAll.h"
#include "../require.h"
using namespace std;

string& stripHTMLTags(string& s) throw(runtime_error) {
    size_t leftPos;
    while((leftPos = s.find('<')) != string::npos) {
        size_t rightPos = s.find('>', leftPos+1);
        if(rightPos == string::npos) {
            ostringstream msg;
            msg << "Incomplete HTML tag starting in position "
                << leftPos;
            throw runtime_error(msg.str());
        }
        s.erase(leftPos, rightPos - leftPos + 1);
    }
    // Remove all special HTML characters
    replaceAll(s, "&lt;", "<");
    replaceAll(s, "&gt;", ">");
    replaceAll(s, "&amp;", "&");
    replaceAll(s, "&nbsp;", " ");
    // Etc...
    return s;
}

```



```

}

int main(int argc, char* argv[]) {
    requireArgs(argc, 1,
        "usage: HTMLStripper2 InputFile");
    ifstream in(argv[1]);
    assure(in, argv[1]);
    // Read entire file into string; then strip
    ostringstream ss;
    ss << in.rdbuf();
    try {
        string s = ss.str();
        cout << stripHTMLTags(s) << endl;
        return EXIT_SUCCESS;
    } catch(runtime_error& x) {
        cout << x.what() << endl;
        return EXIT_FAILURE;
    }
} //::~~

```

在这个程序中，通过把文件流的`rdbuf()`调用放到一个`ostringstream`对象中，最终将整个文件读到一个字符串中。这样搜索HTML文件中的界定符对并将其删除就很容易了，也不必像第3章中的前一版本那样考虑跨越多行的标记。

下面的例子说明了如何使用双向（即，读/写）字符串流：

```

//: C04:StringSeeking.cpp {-bor}{-dmc}
// Reads and writes a string stream.
#include <cassert>
#include <sstream>
#include <string>
using namespace std;

int main() {
    string text = "We will hook no fish";
    stringstream ss(text);
    ss.seekp(0, ios::end);
    ss << " before its time.";
    assert(ss.str() ==
        "We will hook no fish before its time.");
    // Change "hook" to "ship"
    ss.seekg(8, ios::beg);
    string word;
    ss >> word;
    assert(word == "hook");
    ss.seekp(8, ios::beg);
    ss << "ship";
    // Change "fish" to "code"
    ss.seekg(16, ios::beg);
    ss >> word;
    assert(word == "fish");
    ss.seekp(16, ios::beg);
    ss << "code";
    assert(ss.str() ==
        "We will ship no code before its time.");
    ss.str("A horse of a different color.");
    assert(ss.str() == "A horse of a different color.");
} //::~~

```

同前面一样，通过调用`seekp()`移动写指针，调用`seekg()`重定位读指针。字符串流比文件流使用起来更容易，因为任何时候都可以从流的读状态切换到写状态或从写状态切换到读

状态,但是在这个例子中没有引入这方面的内容。在这个过程中,不需要重定位读指针或写指针,或刷新流。在这个程序中也说明了重载的**str()**函数,它用一个新字符串替换流中内置的**stringbuf**。

## 4.8 输出流的格式化

输入输出流的设计目标是使用户易于移动和/或格式化字符。如果不能用C语言中提供的**printf()**及相关函数完成大部分格式化操作,输入输出流将不会有多大用处。在这部分,读者可以学习输入输出流类所有的格式化输出函数,之后可以按照自己的意愿格式化输出数据。

开始学习输入输出流的格式化输出函数的时候可能会引起混淆,因为存在不止一种控制格式化输出的方法:通过成员函数和运算符。之后可能遇到的会引起混淆的地方有:设置状态标志来控制格式化的一般成员函数,如左对齐或右对齐;在十六进制表示法中使用大写字母;始终使用小数点表示浮点数的值等。另一方面,个别的成员函数用来设置和读取填充字符、域宽和精度的值。

为对这些函数进行分类,首先应检查输入输出流的内部格式化数据,以及能够修改这些数据的成员函数。(如果想这样做的话,用成员函数就可以进行所有的操作)。操作符将单独讨论。

### 4.8.1 格式化标志

类**ios**包含一些数据成员,用于存储与流有关的所有格式化信息。其中一些数据存储在变量中,其值有一个变动范围:浮点数精度、输出域宽和用于填充输出的字符(一般为空格)。有关格式化的其他信息由格式化标志决定,为了节省空间,通常多个标志组合在一起使用。成员函数**ios::flags()**可以得到格式化标志所代表的值,这个函数没有参数,函数的返回值为包含当前格式化标志的**fmtflags**(一般被认为是**long**的同义词)对象。其他函数用于改变格式化标志并返回格式化标志的原有值。

```
fmtflags ios::flags(fmtflags newflags);
fmtflags ios::setf(fmtflags ored_flag);
fmtflags ios::unsetf(fmtflags clear_flag);
fmtflags ios::setf(fmtflags bits, fmtflags field);
```

第1个函数强制改变程序需要的所有标志。但更多的时候,是使用其他3个函数,每次改变一个标志。

函数**setf()**的使用似乎会引起一些混淆。要想知道该使用那个版本的重载函数,就需要知道将要改变的格式化标志的类型。有两种类型的格式化标志:一类是仅被设置为开或关的开/关标志;另一类标志和其他的标志联合使用。开/关标志最简单且容易理解,使用**setf(fmtflags)**函数打开标志,使用**unsetf(fmtflags)**函数关闭标志。这些标志在下面的表中说明:

开/关标志	作 用
<b>ios::skipws</b>	跳过空格(输入流的默认情况。)
<b>ios::showbase</b>	打印整型值时指出数字的基数(比如,为 <b>dec</b> 、 <b>oct</b> 或 <b>hex</b> )
	<b>showbase</b> 标志处于开状态时,输入流也能识别前缀基数
<b>ios::showpoint</b>	显示浮点值的小数点并截断数字末尾的零
<b>ios::uppercase</b>	显示十六进制值时使用大写 <b>A~F</b> ,显示科学计数中的 <b>E</b>
<b>ios::showpos</b>	显示正数前的加号(+)
<b>ios::unitbuf</b>	“单元缓冲区”(unit buffering)。每次插入后刷新流



例如，为了在`cout`中显示正号，可以使用语句`cout.setf(ios::showpos)`。如果使用了`cout.unsetf(ios::showpos)`，则不再显示正号。

`unitbuf`标志控制着单元缓冲区（unit buffering），这意味着每次在输出流中插入数据后都会立即刷新该输出流。这对于错误跟踪很方便，当程序崩溃时，数据仍然会保存到日志文件中。下面的程序演示了单元缓冲区：

```
//: C04:Unitbuf.cpp {RunByHand}
#include <cstdlib> // For abort()
#include <fstream>
using namespace std;

int main() {
    ofstream out("log.txt");
    out.setf(ios::unitbuf);
    out << "one" << endl;
    out << "two" << endl;
    abort();
} ///:~
```

在插入任何数据到流对象之前都需要打开单元缓冲区。当把`setf()`注释掉（comment out）时，某个特殊的编译器仅写入了一个字母'o'到文件`log.txt`。而使用单元缓冲区则不会丢失任何数据。

默认情况下，标准错误输出流`cerr`的单元缓冲区处于打开状态。使用单元缓冲会导致大量的系统开销，所以如果程序中频繁使用输出流，则不要使用单元缓冲区，除非不需要考虑程序的执行效率。

#### 4.8.2 格式化域

第2类格式化标志是分组使用的。一次只能设置这些标志中的一个，就像老式汽车收音机上的按钮一样——当按下其中一个按钮时，其他按钮会弹起来。遗憾的是，标志的这种互斥设置不能自动地完成，编程人员必须注意将要设置什么标志，以防错误地使用了`setf()`函数。例如，对于每一种基数（number base）都有相应的标志：十六进制（hexadecimal）、十进制（decimal）和八进制（octal）。这些标志统称为`ios::basefield`。如果已经设置了`ios::dec`标志这时又调用`setf(ios::hex)`，会设置`ios::hex`标志却不会清除`ios::dec`标志位，从而导致不确定的行为。作为前一函数的替代，可以调用第2种形式的`setf()`函数`setf(ios::hex, ios::basefield)`。这个函数首先清除`ios::basefield`域中的所有位，然后设置`ios::hex`标志。这种形式的`setf()`在设置一种标志的时候会确保同组的其他标志被“清除”。`ios::hex`操纵算子自动地完成所有工作，因此不需要关心类的内部实现细节，甚至不需要关心`ios::basefield`是一个二进制标志的集合。接下来，读者将会看到多种操纵算子，在所有使用`setf()`的地方提供相同的功能。

下面是这些标志及其作用：

<code>ios::basefield</code>	作 用
<code>ios::dec</code>	使整型数值的基数为10（十进制形式）（默认的基数——没有前缀）
<code>ios::hex</code>	使整型数值的基数为16（十六进制形式）
<code>ios::oct</code>	使整型数值的基数为8（八进制形式）

<b>ios::floatfield</b>	作 用
<b>ios::scientific</b>	以科学计数法形式显示浮点数。精度域指示小数点后数字的位数
<b>ios::fixed</b>	以固定格式显示浮点数。精度域指示小数点后数字的位数
<b>"automatic"</b> (不设置任何标志位)	精度域指示所有有效数字的位数
<b>ios::adjustfield</b>	作 用
<b>ios::left</b>	使数值左对齐。使用填充字符填充右边空位
<b>ios::right</b>	使数值右对齐。使用填充字符填充左边空位。此为默认对齐方式
<b>ios::internal</b>	把填充字符放到前导正负号或基数指示符之后, 数值之前 (换句话说, 如果输出正负号, 则正负号左对齐, 而数值右对齐。)

#### 4.8.3 宽度、填充和精度设置

用来控制输出域(字段)的宽度、填补输出域的填充字符和显示浮点数精度的内部变量, 由与其同名的成员函数进行读写。

函 数	作 用
<b>int ios::width( )</b>	返回当前宽度。默认为0。用于插入符和提取符
<b>int ios::width(int n)</b>	设定宽度, 并返回先前的宽度
<b>int ios::fill( )</b>	返回当前填充字符。默认为空格符
<b>int ios::fill(int n)</b>	设定填充字符, 并返回先前的填充字符
<b>int ios::precision( )</b>	返回当前浮点数精度。默认情况下, 精确到小数点后6位
<b>int ios::precision(int n)</b>	设定浮点数精度, 返回先前的精度。“precision (精度)”的含义参看 <b>ios::floatfield</b> 表

**fill**和**precision**的值是相当直观的, 但**width**的值就需要解释一下。当宽度为0时, 在流中插入一个值的结果是, 生成表示这个值所需的最少字符数。宽度为正的意思是, 在流中插入一个数, 至少会产生宽度所规定数量的字符; 如果插入字符的个数小于宽度值, 则用填充字符填充空余位置。然而, 输出的值永远都不会被截断, 所以如果试图在宽度为2时打印123, 仍会得到123。宽度只能指定输出字符的最小数目; 没有设定输出字符最大数目的方法。

宽度还有一个明显的不同, 因为每个插入符或提取符都可能受它的值的影响, 所以它被每个插入符或提取符隐含自动重置为0。它不是一个真正的状态变量, 而是插入符和提取符的隐含参数。如果想得到固定不变的宽度, 需要在每次使用插入符或提取符之后调用**width( )**。

#### 4.8.4 一个完整的例子

为了使读者明白如何使用前面讨论过的所有函数, 这里给出一个调用了所有这些函数的例子:

```
//: C04:Format.cpp
// Formatting Functions.
#include <fstream>
#include <iostream>
#include "../require.h"
using namespace std;
#define D(A) T << #A << endl; A

int main() {
    ofstream T("format.out");
    assure(T);
    D(int i = 47;)
```

```

D(float f = 2300114.414159;)
const char* s = "Is there any more?";
D(T.setf(ios::unitbuf);)
D(T.setf(ios::showbase);)
D(T.setf(ios::uppercase | ios::showpos);)
D(T << i << endl;) // Default is dec
D(T.setf(ios::hex, ios::basefield);)
D(T << i << endl;)
D(T.setf(ios::oct, ios::basefield);)
D(T << i << endl;)
D(T.unsetf(ios::showbase);)
D(T.setf(ios::dec, ios::basefield);)
D(T.setf(ios::left, ios::adjustfield);)
D(T.fill('0');)
D(T << "fill char: " << T.fill() << endl;)
D(T.width(10);)
T << i << endl;
D(T.setf(ios::right, ios::adjustfield);)
D(T.width(10);)
T << i << endl;
D(T.setf(ios::internal, ios::adjustfield);)
D(T.width(10);)
T << i << endl;
D(T << i << endl;) // Without width(10)

D(T.unsetf(ios::showpos);)
D(T.setf(ios::showpoint);)
D(T << "prec = " << T.precision() << endl;)
D(T.setf(ios::scientific, ios::floatfield);)
D(T << endl << f << endl;)
D(T.unsetf(ios::uppercase);)
D(T << endl << f << endl;)
D(T.setf(ios::fixed, ios::floatfield);)
D(T << f << endl;)
D(T.precision(20);)
D(T << "prec = " << T.precision() << endl;)
D(T << endl << f << endl;)
D(T.setf(ios::scientific, ios::floatfield);)
D(T << endl << f << endl;)
D(T.setf(ios::fixed, ios::floatfield);)
D(T << f << endl;)

D(T.width(10);)
T << s << endl;
D(T.width(40);)
T << s << endl;
D(T.setf(ios::left, ios::adjustfield);)
D(T.width(40);)
T << s << endl;
} ///:~

```

这个例子中用了一种技巧来创建一个跟踪文件，以监视程序执行时发生了什么事情。宏 **D(a)** 用预处理器（程序）把 **a** 转换成串并显示。然后对 **a** 进行重复迭代，所以语句顺次执行。宏把所有信息输出到跟踪文件 **T**。程序的输出为：

```

int i = 47;
float f = 2300114.414159;
T.setf(ios::unitbuf);
T.setf(ios::showbase);
T.setf(ios::uppercase | ios::showpos);
T << i << endl;
+47

```

```

T.setf(ios::hex, ios::basefield);
T << i << endl;
0X2F
T.setf(ios::oct, ios::basefield);
T << i << endl;
057
T.unsetf(ios::showbase);
T.setf(ios::dec, ios::basefield);
T.setf(ios::left, ios::adjustfield);
T.fill('0');
T << "fill char: " << T.fill() << endl;
fill char: 0
T.width(10);
+470000000
T.setf(ios::right, ios::adjustfield);
T.width(10);
0000000+47
T.setf(ios::internal, ios::adjustfield);
T.width(10);
+000000047
T << i << endl;
+47
T.unsetf(ios::showpos);
T.setf(ios::showpoint);
T << "prec = " << T.precision() << endl;
prec = 6
T.setf(ios::scientific, ios::floatfield);
T << endl << f << endl;

2.300114E+06
T.unsetf(ios::uppercase);
T << endl << f << endl;

2.300114e+06
T.setf(ios::fixed, ios::floatfield);
T << f << endl;
2300114.500000
T.precision(20);
T << "prec = " << T.precision() << endl;
prec = 20
T << endl << f << endl;

2300114.500000000000000000000000
T.setf(ios::scientific, ios::floatfield);
T << endl << f << endl;

2.3001145000000000000000e+06
T.setf(ios::fixed, ios::floatfield);
T << f << endl;
2300114.5000000000000000000000
T.width(10);
Is there any more?
T.width(40);
00000000000000000000000000000000Is there any more?
T.setf(ios::left, ios::adjustfield);
T.width(40);
Is there any more?00000000000000000000000000000000

```

研究这个输出文件可以使读者对输入输出流的格式化成员函数理解得更清晰、明确。



## 4.9 操纵算子

从前面的程序可以看出，调用成员函数进行流的格式化操作有些冗长乏味。为使读操作和写操作更容易，C++语言提供了操纵算子的集合，这些操纵算子可以实现与相应的成员函数相同的功能。操纵算子使用起来更方便，因为可以在一个表达式中插入它们；不需要单独的函数调用语句。

操纵算子改变流的状态而不是（或同时）处理数据。例如，当在一个输出表达式中插入**endl**时，不但在流中插入了一个换行符，而且刷新了流（即，将存储在流内部缓冲区中但还未真正输出的所有字符输出）。也可像这样刷新流：

```
cout << flush;
```

这引起调用成员函数**flush()**的副作用，即

```
cout.flush();
```

（没在流中插入任何东西。）其他的基本操纵算子改变数的基数为**oct**（八进制）、**dec**（十进制）或**hex**（十六进制）：

```
cout << hex << "0x" << i << endl;
```

既然如此，以后的数字输出将继续保持为十六进制模式，直至修改它。通过在输出流中插入**dec**或**oct**来改变这种模式。

也存在一种针对提取操作的操纵算子，它可以“吃掉”空格：

```
cin >> ws;
```

不带参数的操纵算子在头文件**<iostream>**中定义。包括**dec**、**oct**和**hex**，分别对应于**setf(ios::dec, ios::basefield)**、**setf(ios::oct, ios::basefield)**和**setf(ios::hex, ios::basefield)**，但前者更简洁。在头文件**<iostream>**中也包含**ws**、**endl**和**flush**以及在这里说明的其他操纵算子：

操纵算子	作 用
<b>showbase</b>	输出整型数时显示数字的基数（ <b>dec</b> 、 <b>oct</b> 或 <b>hex</b> ）
<b>noshowbase</b>	
<b>showpos</b>	显示正数前面的正号（+）
<b>noshowpos</b>	
<b>uppercase</b>	用大写的A~F显示十六进制数，在科学计数型数字中使用大写的E
<b>nouppercase</b>	
<b>showpoint</b>	显示浮点数的十进制小数点和尾部的0
<b>noshowpoint</b>	
<b>skipws</b>	跳过输入中的空格
<b>noskipws</b>	
<b>left</b>	左对齐，向右边填充字符
<b>right</b>	右对齐，向左边填充字符
<b>internal</b>	把填充字符放到引导符或基数指示符和数值之间
<b>scientific</b>	
<b>fixed</b>	

### 4.9.1 带参数的操纵算子

有6个标准的带参数的操纵算子，如**setw()**。这些操纵算子在头文件**<iomanip>**中定义，

下表对这些操纵算子做了总结：

操纵算子	作用
<b>setiosflags(fmtflags n)</b>	其作用相当于调用函数 <b>setf(n)</b> 。设置后会一直起作用，直到下一次设置将其改变，如调用 <b>ios::setf()</b>
<b>resetiosflags(fmtflags n)</b>	清除由 <b>n</b> 代表的格式化标志。本次设置一直有效，直到进行下一次设置，如调用 <b>ios::unsetf()</b>
<b>setbase(base n)</b>	将数的基数设为 <b>n</b> ， <b>n</b> 的取值为10、8或16。（如传递给 <b>n</b> 的值为其他值则自动置为0）如果 <b>n</b> 的值为0，则输出数的基数为10，但输入使用C语言惯用法：10为10，010为8，0xf为15。对输出流也可使用 <b>dec</b> 、 <b>oct</b> 和 <b>hex</b>
<b>setfill(char n)</b>	将填充字符设为 <b>n</b> ，就像 <b>ios::fill()</b>
<b>setprecision(int n)</b>	将数字精度设为 <b>n</b> ，就像 <b>ios::precision()</b>
<b>setw(int n)</b>	将宽度设为 <b>n</b> ，就像 <b>ios::width()</b>

如果程序中大量使用流格式化操作，则可以发现使用操纵算子代替调用流类成员函数可以简化代码。这里的例子用操纵算子重写了前面的程序。（程序中删除了**D()**宏，使得代码更易阅读。）

```
//: C04:Manips.cpp
// Format.cpp using manipulators.
#include <fstream>
#include <iomanip>
#include <iostream>
using namespace std;

int main() {
    ofstream trc("trace.out");
    int i = 47;
    float f = 2300114.414159;
    char* s = "Is there any more?";

    trc << setiosflags(ios::unitbuf
        | ios::showbase | ios::uppercase
        | ios::showpos);
    trc << i << endl;
    trc << hex << i << endl
        << oct << i << endl;
    trc.setf(ios::left, ios::adjustfield);
    trc << resetiosflags(ios::showbase)
        << dec << setfill('0');
    trc << "fill char: " << trc.fill() << endl;
    trc << setw(10) << i << endl;
    trc.setf(ios::right, ios::adjustfield);
    trc << setw(10) << i << endl;
    trc.setf(ios::internal, ios::adjustfield);
    trc << setw(10) << i << endl;
    trc << i << endl; // Without setw(10)

    trc << resetiosflags(ios::showpos)
        << setiosflags(ios::showpoint)
        << "prec = " << trc.precision() << endl;
    trc.setf(ios::scientific, ios::floatfield);
    trc << f << resetiosflags(ios::uppercase) << endl;
    trc.setf(ios::fixed, ios::floatfield);
    trc << f << endl;
    trc << f << endl;
```



```

trc << setprecision(20);
trc << "prec = " << trc.precision() << endl;
trc << f << endl;
trc.setf(ios::scientific, ios::floatfield);
trc << f << endl;
trc.setf(ios::fixed, ios::floatfield);
trc << f << endl;
trc << f << endl;

trc << setw(10) << s << endl;
trc << setw(40) << s << endl;
trc.setf(ios::left, ios::adjustfield);
trc << setw(40) << s << endl;
} ///:~

```

读者可以看到,在这个程序中有多处地方,将多条语句合并到一条链式表达的插入语句中。注意对函数**setiosflags()**的调用,其参数为几个格式化标志的按位或。前一例子中相同的功能由**setf()**和**unsetf()**实现。

对输出流使用函数**setw()**时,输出表达式被格式化输出到一个临时串,格式化串的长度与传递给**setw()**的参数相比较,根据比较结果决定是否需要用当前填充字符填补空余位置。换言之,**setw()**影响格式化输出操作的结果字符串。同样,对输入流使用**setw()**函数进行设置,只在读字符串时有意义,下面的例子很清楚地说明了这一点:

```

//: C04:InputWidth.cpp
// Shows limitations of setw with input.
#include <cassert>
#include <cmath>
#include <iomanip>
#include <limits>
#include <sstream>
#include <string>
using namespace std;

int main() {
    istringstream is("one 2.34 five");
    string temp;
    is >> setw(2) >> temp;
    assert(temp == "on");
    is >> setw(2) >> temp;
    assert(temp == "e");
    double x;
    is >> setw(2) >> x;
    double relerr = fabs(x - 2.34) / x;
    assert(relerr <= numeric_limits<double>::epsilon());
} ///:~

```

如果试图读一个字符串,函数**setw()**准确地控制着提取字符的数目,读取过程遇到小数点时结束。第1次提取获得了两个字符,而第2次提取仅获得了一个字符,尽管将读取数目设置为两个。这是因为**operator>>()**使用空格作为界定符(除非关闭**skipws**标志)。然而,当试图读取一个数字时,例如读取**x**,不能用**setw()**来限定读取字符的个数。对于输入流,**setw()**只能用于字符串的提取。

#### 4.9.2 创建操纵算子

有时,读者喜欢创建自己的操纵算子,而且创建过程也相当简单。不带参数(零参数, zero-argument)的操纵算子是仅一个函数,例如**endl**,它只是以**ostream**对象的引用为参数,

返回值为一个**ostream**对象的引用的函数。**endl**的声明为：

```
ostream& endl(ostream&);
```

现在，当执行语句

```
cout << "howdy" << endl;
```

时，**endl**将产生该函数的地址。编译器会问，“是否存在一个函数，它以一个函数的地址作为参数？”头文件**<iostream>**中的预定义函数负责这项工作；这些函数称为应用算子（applicator）（因为它们将一个函数应用到流类）。应用算子调用它的函数参数，并传递**ostream**对象作为自己的参数。在这里，不需要知道应用算子是如何创建操纵算子的；仅需知道操纵算子的存在。这里有一个（简化的）**ostream**应用算子的代码：

```
ostream& ostream::operator<<(ostream& (*pf)(ostream&)) {
    return pf(*this);
}
```

实际的定义因涉及模板会更复杂一些，这行代码说明了这项技术。当一个函数，如 **\*pf**（以流作为参数，返回流的引用），被插入到一个流中时，调用上面的应用算子函数，之后执行**pf**指针指向的函数。**ios\_base**、**basic\_ios**、**basic\_ostream**和**basic\_istream**的应用算子在标准C++库中预定义。

这里有一个比较简明的例子解释了上面所描述的过程，例子中创建了一个叫做**nl**的操纵算子，它的作用是在流中插入换行符（也就是说，这个操纵算子不刷新流，不像**endl**那样）：

```
//: C04:nl.cpp
// Creating a manipulator.
#include <iostream>
using namespace std;

ostream& nl(ostream& os) {
    return os << '\n';
}

int main() {
    cout << "newlines" << nl << "between" << nl
        << "each" << nl << "word" << nl;
} ///:~
```

当插入**nl**到一个输出流如**cout**时，调用顺序为：

```
cout.operator<<(nl) → nl(cout)
```

表达式

```
os << '\n';
```

在函数**nl()**内部调用**ostream::operator(char)**，它返回一个流对象，这个流对象最终从**nl()**返回。<sup>①</sup>

#### 4.9.3 效用算子

读者已经看到，零参数的操纵算子很容易创建。但是如何创建带参数的操纵算子呢？如果研究头文件**<iomanip>**，就会发现一种称作**smanip**的类型，它返回带参数的操纵算子。读者也许试图仿照**smanip**定义自己的带参数的操纵算子，但是请不要这样做。因为类型

<sup>①</sup> 在把**nl**定义到头文件之前，使之成为**inline**（内联）函数。



**smanip**是依赖于系统实现的，所以不具备可移植性。幸运的是，可以使用由Jerry Schwarz提出的叫做效用算子（effector）的技术直接定义独立于机器实现的操纵算子。<sup>①</sup>一个效用算子是一个简单的类，该类的构造函数可以格式化一个字符串，这个字符串描述了读者希望的操作，并将这个字符串连同重载的**operator<<**一起插入到流中。这里有一个含有两个效用算子的程序例子。第1个效用算子输出一个截断的字符串，第2个效用算子以二进制方式输出一个数。

```

//: C04:Effector.cpp
// Jerry Schwarz's "effectors."
#include <cassert>
#include <limits> // For max()
#include <sstream>
#include <string>
using namespace std;

// Put out a prefix of a string:
class Fixw {
    string str;
public:
    Fixw(const string& s, int width) : str(s, 0, width) {}
    friend ostream& operator<<(ostream& os, const Fixw& fw) {
        return os << fw.str;
    }
};

// Print a number in binary:
typedef unsigned long ulong;

class Bin {
    ulong n;
public:
    Bin(ulong nn) { n = nn; }
    friend ostream& operator<<(ostream& os, const Bin& b) {
        const ulong ULMAX = numeric_limits<ulong>::max();
        ulong bit = ~(ULMAX >> 1); // Top bit set
        while(bit) {
            os << (b.n & bit ? '1' : '0');
            bit >>= 1;
        }
        return os;
    }
};

int main() {
    string words = "Things that make us happy, make us wise";
    for(int i = words.size(); --i >= 0;) {
        ostringstream s;
        s << Fixw(words, i);
        assert(s.str() == words.substr(0, i));
    }
    ostringstream xs, ys;
    xs << Bin(0xCAFEBAEUL);
    assert(xs.str() ==
        "1100" "1010" "1111" "1110" "1011" "1010" "1011" "1110");
    ys << Bin(0x76543210UL);
    assert(ys.str() ==
        "0111" "0110" "0101" "0100" "0011" "0010" "0001" "0000");
} ///:~

```

① Jerry Schwarz是输入输出流的设计者。

类**Fixw**的构造函数创建**char\***参数的一个被截短的拷贝，由析构函数释放创建拷贝时分配的内存。重载运算符**operator<<**把第2个参数的内容即**Fixw**对象插入到第1个参数即**ostream**对象中，然后返回**ostream**对象，所以它能够在一个链式表达式中使用。当在一个表达式中使用**Fixw**时，如下所示：

```
cout << Fixw(string, i) << endl;
```

该语句调用类**Fixw**的构造函数创建一个**Fixw**临时对象，这个临时对象被传给**operator<<**。它的作用相当于带参数的操纵算子。临时**Fixw**对象在这条语句结束前将一直存在。

**Bin**效用算子依赖这样一个事实：右移无符号数字时在二进制数的高位补零。可以使用**numeric\_limits<unsigned long>::max()**（产生**unsigned long**数的最大值，在标准头文件**<limits>**中定义）利用高位集产生一个值，并且这个值从头至尾进行位移用来询问被测试的数字（通过右移），依次屏蔽每一位。为了具有可读性，现在已经将代码中的字符串文字并列；编译器会将分开的这些字符串合并到一个字符串中。

这项技术历来存在的问题是：一旦为**char\***创建了**Fixw**对象或为**unsigned long**创建了**Bin**对象，就不允许再为**Fixw**类或**Bin**类创建不同的类对象。然而，使用名字空间后这个问题就不存在了。效用算子和操纵算子并不等同，尽管它们可以用来解决相同的问题。如果发现某个问题使用效用算子不能解决，就需要克服操纵算子的复杂性。

## 4.10 输入输出流程序举例

这部分将介绍几个例子，这些例子使用了本章中讲述的知识。尽管存在很多处理字节的工具（UNIX下的流编辑器，如**sed**和**awk**或许是最常用的，而一个文本编辑器也属于此类），但一般来说这些工具有些局限性。**sed**和**awk**可能比较慢，而且只能处理前向序列里的行，并且文本编辑器通常需要人机交互，或至少学习一门专用的宏语言。使用输入输出流编写的程序没有这些限制：具有快速性、可移植性和灵活性。

### 4.10.1 维护类库的源代码

一般来说，当创建一个类时，读者往往会想到有关库的术语：类的声明在头文件**Name.h**中定义，而类的成员函数的实现在文件**Name.cpp**中建立。这些文件有特殊的需求：一个特殊的编码标准（这里的程序使用本教材中的代码格式），而且头文件中的预处理语句能避免类的重复声明。（重复声明使编译器不知道哪个类是程序真正需要的。这些类可能不同，所以编译器会输出一个错误信息。）

这个例子创建一个新的头文件/实现文件对，或修改已存在的一个头文件/实现文件对。如果文件已经存在，则对文件进行检查并修改，如果文件不存在则使用合适的格式创建该文件。

```
//: C04:Cppcheck.cpp
// Configures .h & .cpp files to conform to style
// standard. Tests existing files for conformance.
#include <fstream>
#include <sstream>
#include <string>
#include <cstdint>
#include "../require.h"
using namespace std;

bool startsWith(const string& base, const string& key) {
    return base.compare(0, key.size(), key) == 0;
}
```

```

void cppCheck(string fileName) {
    enum bufs { BASE, HEADER, IMPLEMENT, HLINE1, GUARD1,
        GUARD2, GUARD3, CPPLINE1, INCLUDE, BUFNUM };
    string part[BUFNUM];
    part[BASE] = fileName;
    // Find any '.' in the string:
    size_t loc = part[BASE].find('.');
    if(loc != string::npos)
        part[BASE].erase(loc); // Strip extension
    // Force to upper case:
    for(size_t i = 0; i < part[BASE].size(); i++)
        part[BASE][i] = toupper(part[BASE][i]);
    // Create file names and internal lines:
    part[HEADER] = part[BASE] + ".h";
    part[IMPLEMENT] = part[BASE] + ".cpp";
    part[HLINE1] = "//" " " + part[HEADER];
    part[GUARD1] = "#ifndef " + part[BASE] + "_H";
    part[GUARD2] = "#define " + part[BASE] + "_H";
    part[GUARD3] = "#endif // " + part[BASE] + "_H";
    part[CPPLINE1] = string("//") + " " + part[IMPLEMENT];
    part[INCLUDE] = "#include \"" + part[HEADER] + "\"";
    // First, try to open existing files:
    ifstream existh(part[HEADER].c_str()),
        existcpp(part[IMPLEMENT].c_str());
    if(!existh) { // Doesn't exist; create it
        ofstream newheader(part[HEADER].c_str());
        assure(newheader, part[HEADER].c_str());
        newheader << part[HLINE1] << endl
            << part[GUARD1] << endl
            << part[GUARD2] << endl << endl
            << part[GUARD3] << endl;
    } else { // Already exists; verify it
        stringstream hfile; // Write & read
        ostringstream newheader; // Write
        hfile << existh.rdbuf();
        // Check that first three lines conform:
        bool changed = false;
        string s;
        hfile.seekg(0);
        getline(hfile, s);
        bool lineUsed = false;
        // The call to good() is for Microsoft (later too):
        for(int line = HLINE1; hfile.good() && line <= GUARD2;
            ++line) {
            if(startsWith(s, part[line])) {
                newheader << s << endl;
                lineUsed = true;
                if(getline(hfile, s))
                    lineUsed = false;
            } else {
                newheader << part[line] << endl;
                changed = true;
                lineUsed = false;
            }
        }
    }
    // Copy rest of file
    if(!lineUsed)
        newheader << s << endl;
    newheader << hfile.rdbuf();
    // Check for GUARD3
    string head = hfile.str();
    if(head.find(part[GUARD3]) == string::npos) {
        newheader << part[GUARD3] << endl;
    }
}

```



```

        changed = true;
    }
    // If there were changes, overwrite file:
    if(changed) {
        existh.close();
        ofstream newH(part[HEADER].c_str());
        assure(newH, part[HEADER].c_str());
        newH << "//@//\n" // Change marker
              << newheader.str();
    }
}
if(!existcpp) { // Create cpp file
    ofstream newcpp(part[IMPLEMENT].c_str());
    assure(newcpp, part[IMPLEMENT].c_str());
    newcpp << part[CPPLINE1] << endl
           << part[INCLUDE] << endl;
} else { // Already exists; verify it
    stringstream cppfile;
    ostringstream newcpp;
    cppfile << existcpp.rdbuf();
    // Check that first two lines conform:
    bool changed = false;
    string s;
    cppfile.seekg(0);
    getline(cppfile, s);
    bool lineUsed = false;
    for(int line = CPPLINE1;
        cppfile.good() && line <= INCLUDE; ++line) {
        if(startsWith(s, part[line])) {
            newcpp << s << endl;
            lineUsed = true;
            if(getline(cppfile, s))
                lineUsed = false;
        } else {
            newcpp << part[line] << endl;
            changed = true;
            lineUsed = false;
        }
    }
    // Copy rest of file
    if(!lineUsed)
        newcpp << s << endl;
    newcpp << cppfile.rdbuf();
    // If there were changes, overwrite file:
    if(changed) {
        existcpp.close();
        ofstream newCPP(part[IMPLEMENT].c_str());
        assure(newCPP, part[IMPLEMENT].c_str());
        newCPP << "//@//\n" // Change marker
              << newcpp.str();
    }
}
}

int main(int argc, char* argv[]) {
    if(argc > 1)
        cppCheck(argv[1]);
    else
        cppCheck("cppCheckTest.h");
} ///:~

```

首先注意一个有用的函数**startsWith()**，这个函数的名字说明了它的功能——如果函数

的第1个字符串参数的内容以第2个字符串参数的内容开始（即第2个参数为第1个参数的前缀）时，它返回**true**。在查找期待的注释及相关的包含语句时使用这个函数。定义了字符串数组**part**之后，就可使用循环从头至尾对源代码文件中所期待查找的语句序列进行操作。如果源代码文件不存在，则仅将语句写到用已经给出的文件名命名的新文件中。如果文件存在，则每次搜索文件中的一行，并进行校验该行的出现。如果期待查找的语句不存在，则将其插入到源代码文件中。需要特别注意的是，要确保不要遗漏已经存在的行（参看代码中使用布尔变量**lineUsed**的地方）。注意，现在是在对一个已经存在的文件使用**stringstream**对象，所以能够先写文件的内容至该对象，然后再从该对象中读取和搜索信息。

枚举类型**bufs**中的有名枚举常量分别是：**BASE**，用大写字母表示不带扩展名的基本文件名；**HEADER**，头文件名；**IMPLEMENT**，实现文件（扩展名为**cpp**）名；**HLINE1**，头文件中的第1行基本代码；**GUARD1**、**GUARD2**和**GUARD3**，头文件中的“警戒（guard）”行（防止多重包含）；**CPPLINE1**，**cpp**文件中的第1行；**INCLUDE**，**cpp**文件中包含头文件的语句。

如果运行这个程序但不带任何参数，则会创建下面两个文件：

```
// CPPCHECKTEST.h
#ifndef CPPCHECKTEST_H
#define CPPCHECKTEST_H

#endif // CPPCHECKTEST_H

// CPPCHECKTEST.cpp
#include "CPPCHECKTEST.h"
```

（这里省略了双斜线后面第1行注释中的冒号，以免混淆本教材中的代码提取符。在由执行**cppCheck**产生的真正输出中会包含在此省略的冒号。）

通过从文件中删去某些行然后重新执行程序，可以对程序完成的功能进行测试。可以看到每次执行程序后被删除的行会被写回文件。文件被修改后，在文件的第1行会加入字符串“**//@//**”以使读者注意到文件的变化。再次对文件进行处理前需要去掉这行（否则程序**cppcheck**执行时会假定原文件的第1行注释丢失）。

#### 4.10.2 检测编译器错误

本教材中设计的所有代码在编译时都不会有错误发生。代码中会引起编译时错误的行，将用特殊的注释符号“**//!**”进行注释。下面的程序删去了这些特殊的注释，并添加了带有文件编号的注释行。当读者在自己的编译器上编译这些程序时，可能会产生错误信息，对所有文件进行编译时会看到所有文件的文件编号。这个程序在一个特殊文件中附加修改过的行，从而可以很容易地找出任何一个没有产生错误的行。

```
//: C04:Showerr.cpp {RunByHand}
// Un-comment error generators.
#include <cstddef>
#include <cstdlib>
#include <cstdio>
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>
#include "../require.h"
using namespace std;

const string USAGE =
```

```

"usage: showerr filename chapnum\n"
"where filename is a C++ source file\n"
"and chapnum is the chapter name it's in.\n"
"Finds lines commented with //! and removes\n"
"the comment, appending //( #) where # is unique\n"
"across all files, so you can determine\n"
"if your compiler finds the error.\n"
"showerr /r\n"
"resets the unique counter.";

class Showerr {
    const int CHAP;
    const string MARKER, FNAME;
    // File containing error number counter:
    const string ERRNUM;
    // File containing error lines:
    const string ERRFILE;
    stringstream edited; // Edited file
    int counter;
public:
    Showerr(const string& f, const string& en,
            const string& ef, int c)
        : CHAP(c), MARKER("//!"), FNAME(f), ERRNUM(en),
          ERRFILE(ef), counter(0) {}
    void replaceErrors() {
        ifstream infile(FNAME.c_str());
        assure(infile, FNAME.c_str());
        ifstream count(ERRNUM.c_str());
        if(count) count >> counter;
        int linecount = 1;
        string buf;
        ofstream errlines(ERRFILE.c_str(), ios::app);
        assure(errlines, ERRFILE.c_str());
        while(getline(infile, buf)) {
            // Find marker at start of line:
            size_t pos = buf.find(MARKER);
            if(pos != string::npos) {
                // Erase marker:
                buf.erase(pos, MARKER.size() + 1);
                // Append counter & error info:
                ostringstream out;
                out << buf << " // (" << ++counter << " ) "
                    << "Chapter " << CHAP
                    << " File: " << FNAME
                    << " Line " << linecount << endl;
                edited << out.str();
                errlines << out.str(); // Append error file
            }
            else
                edited << buf << "\n"; // Just copy
            ++linecount;
        }
    }
    void saveFiles() {
        ofstream outfile(FNAME.c_str()); // Overwrites
        assure(outfile, FNAME.c_str());
        outfile << edited.rdbuf();
        ofstream count(ERRNUM.c_str()); // Overwrites
        assure(count, ERRNUM.c_str());
        count << counter; // Save new counter
    }
};

```



```

int main(int argc, char* argv[]) {
    const string ERRCOUNT("../errnum.txt"),
        ERRFILE("../errlines.txt");
    requireMinArgs(argc, 1, USAGE.c_str());
    if(argv[1][0] == '/' || argv[1][0] == '-') {
        // Allow for other switches:
        switch(argv[1][1]) {
            case 'r': case 'R':
                cout << "reset counter" << endl;
                remove(ERRCOUNT.c_str()); // Delete files
                remove(ERRFILE.c_str());
                return EXIT_SUCCESS;
            default:
                cerr << USAGE << endl;
                return EXIT_FAILURE;
        }
    }
    if(argc == 3) {
        Showerr s(argv[1], ERRCOUNT, ERRFILE, atoi(argv[2]));
        s.replaceErrors();
        s.saveFiles();
    }
} ///:~

```

读者可以用自己选择的标记替换文件中的标记。

程序从每个文件中每次读入一行，然后从这行的开头逐个字符搜索指定的标记；修改这一行并把它放入错误行列表和字符串流对象**edited**中。当所有的文件处理结束后，关闭文件（到达文件范围末尾），作为输出文件重新打开它，将**edited**中的内容输出到文件中。注意，计数器也被保存到一个外部文件中。在下一次程序执行时，计数器的计数在上次计数值的基础上增加。

#### 4.10.3 一个简单的数据记录器

这个例子说明了一种可以将数据记录到磁盘，然后检索它以便进行处理的方法。程序想要产生一个多点的海洋温度——深度轮廓图。类**DataPoint**用来保存数据：

```

//: C04:DataLogger.h
// Datalogger record layout.
#ifndef DATALOG_H
#define DATALOG_H
#include <ctime>
#include <iosfwd>
#include <string>
using std::ostream;

struct Coord {
    int deg, min, sec;
    Coord(int d = 0, int m = 0, int s = 0)
        : deg(d), min(m), sec(s) {}
    std::string toString() const;
};

ostream& operator<<(ostream&, const Coord&);

class DataPoint {
    std::time_t timestamp; // Time & day
    Coord latitude, longitude;
    double depth, temperature;
public:
    DataPoint(std::time_t ts, const Coord& lat,

```



```

        const Coord& lon, double dep, double temp)
: timestamp(ts), latitude(lat), longitude(lon),
  depth(dep), temperature(temp) {}

  DataPoint() : timestamp(0), depth(0), temperature(0) {}
  friend ostream& operator<<(ostream&, const DataPoint&);
};
#endif // DATALOG_H ///:~

```

类**DataPoint**包含一个时间标志，时间标志为头文件<ctime>中定义的**time\_t**类型的值，还有经度和纬度坐标，以及深度和温度值。在这里使用插入符进行格式化操作。下面是文件的实现：

```

//: C04:DataLogger.cpp {0}
// Datapoint implementations.
#include "DataLogger.h"
#include <iomanip>
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

ostream& operator<<(ostream& os, const Coord& c) {
    return os << c.deg << ' ' << c.min << '\n'
        << c.sec << '\n';
}

string Coord::toString() const {
    ostringstream os;
    os << *this;
    return os.str();
}

ostream& operator<<(ostream& os, const DataPoint& d) {
    os.setf(ios::fixed, ios::floatfield);
    char fillc = os.fill('0'); // Pad on left with '0'
    tm* tdata = localtime(&d.timestamp);
    os << setw(2) << tdata->tm_mon + 1 << '\n'
        << setw(2) << tdata->tm_mday << '\n'
        << setw(2) << tdata->tm_year+1900 << '\n'
        << setw(2) << tdata->tm_hour << ':'
        << setw(2) << tdata->tm_min << ':'
        << setw(2) << tdata->tm_sec;
    os.fill(' '); // Pad on left with ' '
    streamsize prec = os.precision(4);
    os << " Lat:" << setw(9) << d.latitude.toString()
        << ", Long:" << setw(9) << d.longitude.toString()
        << ", depth:" << setw(9) << d.depth
        << ", temp:" << setw(9) << d.temperature;
    os.fill(fillc);
    os.precision(prec);
    return os;
} ///:~

```

使用函数**Coord::toString()**是必要的，因为类**DataPoint**的插入符在输出经度和纬度之前调用了**setw()**。无论何时对**Coord**对象使用流插入符，宽度只对第1次插入（即插入数据到**Coord::deg**）有效，因为宽度改变后总是立即重置。调用函数**setf()**使得输出浮点数时精度是固定的，函数**precision()**设置精度为小数点后四位十进制数。请注意，程序中是如何恢复在调用插入符之前设置填充字符和数据精度的。

为得到存储在**DataPoint::timestamp**中的各个测试点的测试时间，可以调用函数**std::localtime()**，该函数返回指向**tm**对象的静态指针。结构**tm**布局（定义）如下：



```

struct tm {
    int tm_sec; // 0-59 seconds
    int tm_min; // 0-59 minutes
    int tm_hour; // 0-23 hours
    int tm_mday; // Day of month
    int tm_mon; // 0-11 months
    int tm_year; // Years since 1900
    int tm_wday; // Sunday == 0, etc.
    int tm_yday; // 0-365 day of year
    int tm_isdst; // Daylight savings?
};

```

### 1. 产生测试数据

这里的程序用来建立一个二进制格式的测试数据文件（使用**write()**函数），使用**DataPoint**插入符建立ASCII格式的第2个文件。也可以把这些数据显示到屏幕上，但以文件形式查看更方便。

```

//: C04:Datagen.cpp
// Test data generator.
//{L} DataLogger
#include <cstdlib>
#include <ctime>
#include <cstring>
#include <fstream>
#include "DataLogger.h"
#include "../require.h"
using namespace std;

int main() {
    time_t timer;
    srand(time(&timer)); // Seed the random number generator
    ofstream data("data.txt");
    assure(data, "data.txt");
    ofstream bindata("data.bin", ios::binary);
    assure(bindata, "data.bin");
    for(int i = 0; i < 100; i++, timer += 55) {
        // Zero to 199 meters:
        double newdepth = rand() % 200;
        double fraction = rand() % 100 + 1;
        newdepth += 1.0 / fraction;
        double newtemp = 150 + rand() % 200; // Kelvin
        fraction = rand() % 100 + 1;
        newtemp += 1.0 / fraction;
        const DataPoint d(timer, Coord(45,20,31),
                           Coord(22,34,18), newdepth,
                           newtemp);
        data << d << endl;
        bindata.write(reinterpret_cast<const char*>(&d),
                      sizeof(d));
    }
} //::~~

```

文件**data.txt**为ASCII格式、采用顺序方法创建的顺序文件，而文件**data.bin**为二进制格式文件，构造函数根据标志**ios::binary**建立此文件。为了说明文本文件采用的格式化形式，这里给出文件**data.txt**的第1行内容（因为行的长度大于本教材页的宽度，所以进行了换行）：

```

07\28\2003 12:54:40 Lat:45*20'31", Long:22*34'18", depth:
16.0164, temp: 242.0122

```

标准C库函数**time()**用执行该语句的当前时间来更新由函数参数指向的**time\_t**的值，在

大多数操作平台上，这个时间是从1970年1月1日00:00:00 GMT (Aquarius (水瓶星座) 时代的黎明?) 开始的秒的计数值。利用标准C中的库函数**srand()**作为随机数产生器来设置当前时间也是很方便的方法，这里就是如此。

之后，把**timer**定时器增加55秒，在各个模拟读操作之间产生有趣的间隔。

各采集点的经度和纬度值采用固定值，表示所采集的数据集是在某个特定的区域。深度和温度值由标准C库函数**rand()**产生，该函数返回一个0到依赖于操作平台的常量**RAND\_MAX**之间的伪随机数，**RAND\_MAX**常量（一般为所在操作平台的无符号整形最大值）在文件**<cstdlib>**中定义。为把得到的伪随机数限制在一个期望的合理范围内，使用取模运算符**%**（从整数相除得到余数）和范围的上限来限定。这些伪随机数都是整数，为了添加小数部分，第2次调用**rand()**以产生小数，将得到的值加1后取倒数（防止除数为0的错误）。

本程序中，文件**data.bin**被用作数据容器，尽管这个数据容器存在于磁盘而不是RAM中。函数**write()**把数据以二进制方式输出到磁盘上。函数的第1个参数是源数据块的起始地址——注意必须将参数设置为**char\***类型，因为函数**write()**使用专用流（narrow stream）。第2个参数是要写出的字符数目，在这个例子中就是**DataPoint**类对象的大小（再一次指明，因为使用的是窄字符流）。因为类**DataPoint**不含指针，所以输出这个类的对象到磁盘上不会产生问题。如果类对象很复杂，则必须实现串行化（serialization）设计，把指针指向的数据写入磁盘，在以后读回数据时再定义新的指针。（不在本卷中讨论串行化——大部分商业化销售的类库都有一些串行化结构来构造它们。）

## 2. 校验和查看数据

为校验以二进制格式存储的数据的正确性，可以用输入流的成员函数**read()**将数据读到内存，然后和最初由**Datagen.cpp**生成的文本文件进行比较。下面的例子仅把格式化的结果输出到**cout**，但读者可以把这些输出重新送到一个文件中，然后用文件比较“实用程序”来进行校验，校验这个文件与最初的文件是否完全相同：

```
//: C04:Datascan.cpp
//{L} DataLogger
#include <fstream>
#include <iostream>
#include "DataLogger.h"
#include "../require.h"
using namespace std;
int main() {
    ifstream bindata("data.bin", ios::binary);
    assure(bindata, "data.bin");
    DataPoint d;
    while(bindata.read(reinterpret_cast<char*>(&d),
        sizeof d))
        cout << d << endl;
} ///:~
```

## 4.11 国际化

现在，软件工业是一个新兴的、健康的、具有世界范围的经济市场，这需要应用程序能在多种语言环境下运行。早在20世纪80年代，C标准委员会就加入了对非美国表达方式习惯的区域性（locale）机制的支持。所谓区域性是在显示一些实体，如时间和货币数量时当地人们习惯使用的方式。在20世纪90年代，C标准委员会同意补充处理宽字符（wide character）（由类型**wchar\_t**表示）的特殊函数进入标准C，容许这些函数支持非ASCII码字符集，一般用于西

欧诸国范围。尽管宽字符所占空间的大小并不特殊，但在一些操作平台上把它按32位字长进行实现，可以满足统一代码协会（Unicode Consortium）对编码的特殊需要，同时也适用于亚洲标准化组织定义的多字节字符集。C++支持宽字符和区域（locale）字符，把两者整合到了输入输出流类库中。

#### 4.11.1 宽字符流

宽字符流（wide stream）是一个处理那些宽字符的流类。目前引入的所有例子（除了第3章中那些带有宽字符特征的例子外）都使用专门处理**char**类型的窄（narrow）字符流。因为流操作的本质都是一样的，与基础字符类型没有关系，所以一般将其封装成模板。例如，可以将所有输入流类都与类模板**basic\_istream**连接来定义：

```
template<class charT, class traits = char_traits<charT> >
class basic_istream {...};
```

事实上，根据下面的类型定义，所有输入流类都是该模板的特化：

```
typedef basic_istream<char> istream;
typedef basic_istream<wchar_t> wistream;
typedef basic_ifstream<char> ifstream;
typedef basic_ifstream<wchar_t> wifstream;
typedef basic_istreamstream<char> istringstream;
typedef basic_istreamstream<wchar_t> wistringstream;
```

其他流类型的定义与此类似。

总而言之，读者用这些方法可以创建不同字符类型的流。但事情也不是那么简单。原因是提供给**char**类型和**wchar\_t**类型的字符处理函数的名称不相同。比较两个窄字符串，比如使用函数**strcmp()**。而用于两个宽字符的比较函数为**wcscmp()**。（请记住这些函数在C语言中的原始声明，这些函数没有重载版本，所有的函数名需要具有惟一性。）正因为如此，一般情况下流类对象的比较运算符不能仅仅调用**strcmp()**。这就需要引入一种方法，使用这种方法可以在进行流对象的比较操作时自动调用正确的底层函数。

解决的办法是找出它们因子的差异成为一个新的抽象。对字符的操作被抽象成为一个**char\_traits**模板，正如在第3章结尾处讨论的，这个模板中预定义了**char**（字符型）和**wchar\_t**（宽字符型）类型。比较两个字符串时，**basic\_string**先调用**traits::compare()**（记住特征参数**traits**是模板的第2个参数），**traits::compare()**再根据所用的字符类型调用**strcmp()**或**wcscmp()**。（对于**basic\_string**来说，这一点是必须清晰的。）

如果访问底层字符处理函数，只需要关注**char\_traits**，但大多数情况下不需要特别注意。然而，为了增强程序的健壮性，需要将插入符和提取符定义为模板，以适应用户要在宽字符流上对它们的使用。

为解释清楚，回忆一下本章开始时引入的类**Date**中的插入符。它的原始定义如下：

```
ostream& operator<<(ostream&, const Date&);
```

这个插入符只能用于窄字符流。为了使其具有通用性，现在把它定义成基于**basic\_ostream**的模板：

```
template<class charT, class traits>
std::basic_ostream<charT, traits>&
operator<<(std::basic_ostream<charT, traits>& os,
          const Date& d) {
    charT fillc = os.fill(os.widen('0'));
    charT dash = os.widen('-');
```

```
os << setw(2) << d.month << dash
    << setw(2) << d.day << dash
    << setw(4) << d.year;
os.fill(fillc);
return os;
}
```

注意，也需要用模板参数**charT**替换**char**来声明**fillc**，因为**fillc**的声明依赖于模板实例化时的参数是**char**还是**wchar\_t**。

因为在定义模板时不知道所使用的流的类型，所以需要有一种自动将字符文字的长度转换成对于该流来说大小合适的方法。成员函数**widen()**负责处理这项工作。例如，对表达式**widen('-')**来说就是将其参数转变成**L'-'**（文字语法相当于**wchar\_t('-')**转变），如果为宽字符流则不进行转换。反之亦然。如果需要，函数**narrow()**将字符转换成**char**类型。

可以使用**widen()**为本章前面较早提供的程序例子编写一个名为**nl**操纵算子的通用版本：

```
template<class charT, class traits>
basic_ostream<charT,traits>&
nl(basic_ostream<charT,traits>& os) {
    return os << charT(os.widen('\n'));
}
```

4.11.2 区域性字符流

不同国家的计算机输出之间最显著的不同，在于分割整数和实数的小数部分所使用的标点符号。在美国，一个句号表示一个小数点，但是世界上大多数国家用逗号表示小数点。如果为各个区域的国家的输出显示分别定义不同的格式，这是十分不方便的。这里再一次使用抽象来解决这个问题。

这次的抽象是区域。每一流类都有相联系的区域对象，这些对象用来指出如何显示不同的文化背景下的确定的数量。一个区域对象管理一系列视文化不同而定的数量的显示规则，定义如下：

种 类	作 用
<b>collate</b>	允许按照不同的比较顺序比较字符串
<b>ctype</b>	对字符类型和 <ctype> 中的惯用程序进行抽象
<b>monetary</b>	支持货币数量的不同格式显示
<b>numeric</b>	支持实数的不同格式的显示，包括数的基（小数点）和分组（每千位）符号
<b>time</b>	支持多种不同格式的时间和日期的显示
<b>messages</b>	其实现依赖于不同内容的消息目录（如不同语言下的错误消息）

下面的程序说明了基本区域性字符流的行为：

```
//: C04:Locale.cpp {-g++}{-bor}{-edg} {RunByHand}
// Illustrates effects of locales.
#include <iostream>
#include <locale>
using namespace std;

int main() {
    locale def;
    cout << def.name() << endl;
    locale current = cout.getloc();
    cout << current.name() << endl;
    float val = 1234.56;
    cout << val << endl;
}
```

```

// Change to French/France
cout.imbue(locale("french"));
current = cout.getloc();
cout << current.name() << endl;
cout << val << endl;

cout << "Enter the literal 7890,12: ";
cin.imbue(cout.getloc());
cin >> val;
cout << val << endl;
cout.imbue(def);
cout << val << endl;
} ///:~

```

输出结果如下:

```

C
C
1234.56
French_France.1252
1234,56
Enter the literal 7890,12: 7890,12
7890,12
7890.12

```

默认的区域为“C”区域,是C和C++程序员多年来一直使用的(基本上是英语和美语文化)。所有的流最初都完全“浸透(imbue)”在“C”区域环境下。成员函数**imbue()**改变了一个流使用的区域。注意程序输出了“法语”区域在ISO(国际标准化组织)中的全称(即在法国表达的“法语”相对于其他国家表达的“法语”)。这个例子说明在这种区域下,数字中的小数点用逗号表示。如果想在一种区域的规则下进行输入,则需要把**cin**改变到相同的区域下工作。

每个区域目录被分成几个领域,每个领域都是一些对应于相应目录封装了特定功能的类。例如,目录**time**包含的领域有**time\_put**和**time\_get**,它们分别含有进行时间、日期的输入(**input**)和输出(**output**)的函数。而目录**monetary**包含的领域有**money\_get**、**money\_put**和**money\_punct**。**money\_punct**决定了流通中的货币符号。)下面的程序说明了**money\_punct**领域。**(time**领域需要用到一种复杂的迭代器,它超出了本章的讨论范围。)

```

//: C04:Facets.cpp {-bor}{-g++}{-mwcc}{-edg}
#include <iostream>
#include <locale>
#include <string>
using namespace std;

int main() {
    // Change to French/France
    locale loc("french");
    cout.imbue(loc);
    string currency =
        use_facet<money_punct<char>>(loc).curr_symbol();
    char point =
        use_facet<money_punct<char>>(loc).decimal_point();
    cout << "I made " << currency << 12.34 << " today!"
        << endl;
} ///:~

```

输出了法国流通货币符号和小数点分隔符:

```
I made €12,34 today!
```



读者也可定义自己的领域以构建个人化的区域。<sup>①</sup>但要当心，区域的开销是相当可观的。事实上，一些供货商提供了区别于标准C++库的不同风格的库，以便满足对使用标准库有限制条件的环境。<sup>②</sup>

## 4.12 小结

本章详细地介绍了输入输出流类库。从本章中学到的内容可以满足读者使用输入输出流创建程序的需要。注意，输入输出流的一些附加的特性并不常用，读者可以查阅输入输出流头文件和阅读编译器文档或本章及附录中提到的参考文献。

## 4.13 练习

- 4-1 有一个由创建的**ifstream**对象打开的文件。创建一个**ostream**对象，使用其成员函数**rddbuf()**读该文件全部内容到**ostream**对象中。提取文件基础缓冲区的**string**拷贝，使用标准C语言头文件<cctype>中定义的宏**toupper()**将每个字符转换为大写。将结果输出到一个新文件。
- 4-2 编写程序：打开一个文件（文件名作为命令行的第1个参数），并搜索文件中单词集合中的任意一个单词（作为参数出现在命令行上）。每次读入一行并将匹配的行（连同行号一起）写到一个新文件中。
- 4-3 编写一个程序：添加“版权声明”到所有源代码文件的开始位置，这些信息通过程序命令行参数指明。
- 4-4 使用自己喜欢的文本搜索程序（如**grep**），输出包含一种特殊模式的所有文件的文件名（仅输出文件名）。重新发送输出到一个新文件。编写一个程序，用这个新文件里的内容来产生一个批处理文件，这个批处理文件对每个由文本搜索程序找到的文件，调用自编的编辑器进行编辑。
- 4-5 我们知道使用**setw()**可以设置读入字符的最小个数，但是如何设置读入字符的最大数量？编写一个效用算子，使得用户可以指定提取字符数目的最大值。该效用算子也可以进行输出。输出时，这个效用算子可以截短输出域的宽度，如果需要可以保持域宽限制的设置。
- 4-6 编程演示如下过程：如果失败或致命错误标志位设置后，随后突然产生流异常，流将立即抛出异常。
- 4-7 由字符串流提供转换很容易实现，不过要付出一定的代价。编写一个程序，实现**stringstream**转换系统，把这个程序和**atoi()**比较，以便观察**stringstream**转换系统最终花费的代价。
- 4-8 编写一个结构**Person**，结构的数据域包含名字，年龄，地址等。其中的字符串类型数据成员为固定大小的数组。每条记录的关键字为身份证号码（社会保险编号）。实现下面的类**Database**：

① 详情请参看 Langer & Kreft 的著作。

② 例如，参看 Dinkumware 的 Abridged 库的网址 <http://www.dinkumware.com>。这个库不支持 locale，对异常的支持为可选项。

```

class DataBase {
public:
    // Find where a record is on disk
    size_t query(size_t ssn);
    // Return the person at rn (record number)
    Person retrieve(size_t rn);
    // Record a record on disk
    void add(const Person& p);
};

```

写一些**Person**记录到磁盘（不要把这些记录都放在内存中）。当用户需要时，从磁盘将这些记录读回到内存。**DataBase**类中的I/O操作使用**read()**和**write()**处理所有**Person**记录。

- 4-9 为结构**Person**编写一个插入符**operator<<**，实现对读入的记录用格式化形式显示。通过将数据输出到文件来演示该插入符的功能。
- 4-10 假定存储结构**Person**的数据库丢失了，但前一个练习中产生的输出文件还存在。使用这个文件重新建立数据库。要确保程序中使用错误检查。
- 4-11 写1 000 000次**size\_t(-1)**（操作平台规定的最大的无符号整数**unsigned int**）到一个文本文件。再以二进制格式写1 000 000次**size\_t(-1)**到一个二进制文件。比较两个文件的大小，看二进制格式的文件能节省多少空间。（读者或许首先想要计算出在自己的操作平台上能节省多少空间。）
- 4-12 打印一个无理数如**sqrt(2.0)**时，通过重复增加函数**precision()**的参数值，观察输入输出流实现的显示该数精度位数的最大个数。
- 4-13 编写一个程序，从文件中读入一些实数，并且显示（打印）这些实数的和、平均值、最小值和最大值。
- 4-14 在执行前，猜测下面程序的输出结果：

```

//: C04:Exercise14.cpp
#include <fstream>
#include <iostream>
#include <sstream>
#include "../require.h"
using namespace std;

#define d(a) cout << #a " ==\t" << a << endl;

void tellPointers(fstream& s) {
    d(s.tellp());
    d(s.tellg());
    cout << endl;
}

void tellPointers(stringstream& s) {
    d(s.tellp());
    d(s.tellg());
    cout << endl;
}

int main() {
    fstream in("Exercise14.cpp");
    assure(in, "Exercise14.cpp");
    in.seekg(10);
    tellPointers(in);
    in.seekp(20);
    tellPointers(in);
    stringstream memStream("Here is a sentence.");
    memStream.seekg(10);
}

```



```

    tellPointers(memStream);
    memStream.seekp(5);
    tellPointers(memStream);
} ///:~

```

#### 4-15 假定要从按如下格式存储数据的文件中按行提取数据:

```

//: C04:Exercise15.txt
Australia
5E56,7667230284,Langler,Tyson,31.2147,0.00042117361
2B97,7586701,Oneill,Zeke,553.429,0.0074673053156065
4D75,7907252710,Nickerson,Kelly,761.612,0.010276276
9F2,6882945012,Hartenbach,Neil,47.9637,0.0006471644
Austria
480F,7187262472,Oneill,Dee,264.012,0.00356226040013
1B65,4754732628,Haney,Kim,7.33843,0.000099015948475
DA1,1954960784,Pascente,Lester,56.5452,0.0007629529
3F18,1839715659,Elsea,Chelsy,801.901,0.010819887645
Belgium
BDF,5993489554,Oneill,Meredith,283.404,0.0038239127
5AC6,6612945602,Parisienne,Biff,557.74,0.0075254727
6AD,6477082,Pennington,Lizanne,31.0807,0.0004193544
4D0E,7861652688,Sisca,Francis,704.751,0.00950906238
Bahamas
37D8,6837424208,Parisienne,Samson,396.104,0.0053445
5E98,6384069,Willis,Pam,90.4257,0.00122009564059246
1462,1288616408,Stover,Hazal,583.939,0.007878970561
5FF3,8028775718,Stromstedt,Bunk,39.8712,0.000537974
1095,3737212,Stover,Denny,3.05387,0.000041205248883
7428,2019381883,Parisienne,Shane,363.272,0.00490155
///:~

```

这些数据按地区划分成若干部分，每部分的开头是一个地区名称，下面的每行都是该地区的每一个销售人员的信息。由逗号分隔开的域（字段）代表每个销售人员的相关数据。每行的第1个域是SELLER\_ID，遗憾的是，这个域是按十六进制数的格式写的。第2个域是PHONE\_NUMBER（注意，有一些域缺少地区编码）。接下来是LAST\_NAME和FIRST\_NAME。TOTAL\_SALES是倒数第2栏。最后一栏是这个销售人员的售货量占公司总售货量的百分比的小数表示。编写程序，在终端窗口用格式化方式显示这些数据，执行结果可以很容易地显示各个销售人员业绩的趋势。输出的样本如下所示：

```

                Australia
-----
*Last Name*  *First Name*  *ID*   *Phone*      *Sales*  *Percent*
Langler      Tyson         24150  766-723-0284  31.24    4.21E-02
Oneill       Zeke          11159  XXX-758-6701  553.43   7.47E-01
(etc.)

```



## 深入理解模板

C++模板应用的便利性远远超出了它只是一种“T类型容器”(containers of T)的范畴。尽管其最初的设计动机是为了能产生类型安全的通用容器，但在现代C++中，模板也用来生成自定义代码，这些代码通过编译时的程序设计构造来优化程序的执行。

本章将从实用的角度来看现代C++中利用模板编程的强大能力（以及缺陷）。对于与模板相关的C++语言的优点和缺陷的更完备的分析，推荐读者阅读由Daveed Vandevoorde和Nico Josuttis<sup>①</sup>所著的那本极棒的书。

### 5.1 模板参数

正如在第1卷中描述的那样，模板有两类：函数模板和类模板。二者都是由它们的参数来完全地描绘模板的特性。每个模板参数描述了下述内容之一：

- 1) 类型（或者是系统固有类型或者是用户自定义类型）。
- 2) 编译时常数值（例如，整数、指针和某些静态实体的引用，通常是作为无类型参数的引用）。
- 3) 其他模板。

第1卷中所举的例子都属于第1种情况，也是最常用的。现在作为简单的类似容器模板的典型示例似乎就是**Stack**类。作为容器，**Stack**对象与容器中存储的对象的类型毫无关联；持有对象的逻辑独立于所持有的对象的类型。基于这个原因，可以用一个类型参数来代表所包含的类型：

```
template<class T> class Stack {
    T* data;
    size_t count;
public:
    void push(const T& t);
    // Etc.
};
```

某个特定的**Stack**实例所使用的实际类型，由参数**T**的实参类型来决定：

```
Stack<int> myStack; // A Stack of ints
```

编译器通过用**int**替代**T**生成相应的代码，从而提供了一个**Stack**的**int**版。在这个例子中，由模板生成类的实例的名字是**Stack<int>**。

#### 5.1.1 无类型模板参数

一个无类型模板参数必须是一个编译时所知的整数值。举个例子，可以创建一个固定长度的**Stack**，指定一个无类型参数作为其中基础数组的大小，如下所示：

```
template<class T, size_t N> class Stack {
    T data[N]; // Fixed capacity is N
    size_t count;
```

① Vandevoorde 和 Josuttis 所著的《C++ Templates: The complete Guide》Addison Wesley, 2003。注意“Daveed”有时会写作“David”。

```
public:
    void push(const T& t);
    // Etc.
};
```

当需要这个模板的一个实例时，必须为参数**N**提供一个编译时常数值，例如：

```
Stack<int, 100> myFixedStack;
```

由于**N**的值在编译时是已知的，内含的数组（**data**）可以被置于运行时堆栈而不是动态存储空间。这种方式避免了与动态内存分配的高层关联，从而提高了运行性能。根据之前提过的模式，上述模板的实例化类名字是**Stack<int, 100>**。这意味着任何一个**N**的不同取值都会产生一个惟一的类类型。例如，**Stack<int, 99>**与**Stack<int, 100>**就是两个不同的类。

将在第7章详细讨论的**bitset**类模板，是标准C++库中惟一使用了无类型模板参数（它指定了**bitset**对象所持有的位的数目）的类。下面的随机数生成器的例子使用了**bitset**来跟踪这些数，这样在随机数生成器下一次工作周期重新开始之前，所有在允许范围内的数都将无重复地按照随机顺序返回。这个例子也重载了运算符**operator()**，用来产生一个熟悉的功能调用语法。

```
//: C05:Urand.h {-bor}
// Unique randomizer.
#ifndef URAND_H
#define URAND_H
#include <bitset>
#include <cstdint>
#include <cstdlib>
#include <ctime>
using std::size_t;
using std::bitset;

template<size_t UpperBound> class Urand {
    bitset<UpperBound> used;
public:
    Urand() { srand(time(0)); } // Randomize
    size_t operator()(); // The "generator" function
};

template<size_t UpperBound>
inline size_t Urand<UpperBound>::operator()() {
    if(used.count() == UpperBound)
        used.reset(); // Start over (clear bitset)
    size_t newval;
    while(used[newval = rand() % UpperBound])
        ; // Until unique value is found
    used[newval] = true;
    return newval;
}
#endif // URAND_H ///:~
```

由**Urand**生成的数全是独一无二的，这是因为**bitset used**跟踪了随机空间中（上限设置成模板参数）所有可能产生的数，并且设置相应的状态位来记录每一个使用过的数。当这些数全都用完了之后，**bitset**被清空以便为下一次工作重新开始做准备。下面是一个描述如何使用**Urand**对象的简单的驱动程序：

```
//: C05:UrandTest.cpp {-bor}
#include <iostream>
#include "Urand.h"
using namespace std;
```

```
int main() {
    Urand<10> u;
    for(int i = 0; i < 20; ++i)
        cout << u() << ' ';
} ///:~
```

正像将在本章后面解释的那样，无类型模板参数在优化数值的计算方面也是很重要的。

### 5.1.2 默认模板参数

在类模板中，可以为模板参数提供默认（缺省）参数，但是在函数模板中却不行。作为默认的模板参数，它们只能被定义一次，编译器会知道第1次的模板声明或定义。一旦引入了一个默认参数，所有它之后的模板参数也必须具有默认值。例如，为了使前面介绍的固定大小的**Stack**模板更友好一些，可以加入一个默认参数，如下所示：

```
template<class T, size_t N = 100> class Stack {
    T data[N]; // Fixed capacity is N
    size_t count;
public:
    void push(const T& t);
    // Etc.
};
```

现在，如果在声明一个**Stack**对象时省略了第2个模板参数，**N**的值将默认为100。

也可以为所有参数提供默认值，但当声明一个实例时必须使用一对空的尖括号，这样编译器就知道说明了一个类模板。

```
template<class T = int, size_t N = 100> // Both defaulted
class Stack {
    T data[N]; // Fixed capacity is N
    size_t count;
public:
    void push(const T& t);
    // Etc.
};
```

```
Stack<> myStack; // Same as Stack<int, 100>
```

默认参数大量用于标准C++库中。比如**vector**类模板声明如下：

```
template<class T, class Allocator = allocator<T> >
class vector;
```

注意，在最后两个右尖括号字符之间有空格。这就避免了编译器将那两个字符(> >)解释为右移运算符。

这个声明说明了**vector**有两个参数：一个参数表示它持有的包含对象的类型，另一个参数代表**vector**所使用的分配器。任何时候只要省略了第2个参数，就会使用标准**allocator**模板，它的参数由第1个模板参数来确定。这个声明也说明，可以在随后的次一级模板的参数中使用该模板参数，就像在这里使用**T**一样。

尽管不能在函数模板中使用默认的模板参数，却能够用模板参数作为普通函数的默认参数。下面的函数模板在参数列表中加入了一个元素：

```
///: C05:FuncDef.cpp
#include <iostream>
using namespace std;

template<class T> T sum(T* b, T* e, T init = T()) {
```

```

while(b != e)
    init += *b++;
    return init;
}

int main() {
    int a[] = { 1, 2, 3 };
    cout << sum(a, a + sizeof a / sizeof a[0]) << endl; // 6
} ///:~

```

**sum()**的第3个参数是作为对这些元素进行累积运算的初始值。由于省略了它，这个参数就默认为是**T()**，在这里是**int**或其他系统固有的类型，它调用了一个伪构造函数执行零初始化操作。

### 5.1.3 模板类型的模板参数

模板可以接受的第3种模板参数类型是另一个类模板。如果想在代码中将一个模板类参数用作另一个模板，编译器首先需要知道这个参数是一个模板。下面的例子说明了一个模板类型的模板参数：

```

//: C05:TempTemp.cpp
// Illustrates a template template parameter.
#include <cstdint>
#include <iostream>
using namespace std;

template<class T>
class Array { // A simple, expandable sequence
    enum { INIT = 10 };
    T* data;
    size_t capacity;
    size_t count;
public:
    Array() {
        count = 0;
        data = new T[capacity = INIT];
    }
    ~Array() { delete [] data; }
    void push_back(const T& t) {
        if(count == capacity) {
            // Grow underlying array
            size_t newCap = 2 * capacity;
            T* newData = new T[newCap];
            for(size_t i = 0; i < count; ++i)
                newData[i] = data[i];
            delete [] data;
            data = newData;
            capacity = newCap;
        }
        data[count++] = t;
    }
    void pop_back() {
        if(count > 0)
            --count;
    }
    T* begin() { return data; }
    T* end() { return data + count; }
};

template<class T, template<class> class Seq>

```



```

class Container {
    Seq<T> seq;
public:
    void append(const T& t) { seq.push_back(t); }
    T* begin() { return seq.begin(); }
    T* end() { return seq.end(); }
};

int main() {
    Container<int, Array> container;
    container.append(1);
    container.append(2);
    int* p = container.begin();
    while(p != container.end())
        cout << *p++ << endl;
} ///:~

```

**Array**类模板是个很平常的序列容器。**Container**模板包含两个参数：一个参数是它持有的类对象类型，还有一个参数是它持有的类对象类型的序列数据结构。在**Container**类的实现中下面一行语句通知编译器，**Seq**是一个模板：

```
Seq<T> seq;
```

如果还没有声明**Seq**是一个模板类型的模板参数，编译器就不会在这里将**Seq**解释为一个模板，尽管已经如此使用了它。在**main()**中使用了一个持有整数的**Array**将一个**Container**实例化，因此本例中的**Seq**代表**Array**。

注意，在本例**Container**的声明中对**Seq**的参数进行命名不是必需的。所讨论的这一行是：

```
template<class T, template<class> class Seq>
```

尽管可以这样写：

```
template<class T, template<class U> class Seq>
```

无论什么地方参数**U**都不是必需的。加上这个参数仅仅是为了说明**Seq**是一个持有单一类型参数的类模板。这种情况类似于某些时候省略函数参数的名称，当不需要它们的时候就可以省略掉。例如当重载自增（增1）运算符++时：

```
T operator++(int);
```

这里的**int**仅仅是一个占位符，并不需要有变量名称。

下面的程序使用了一个固定大小的数组，它有一个特别的模板参数表示数组的长度：

```

///< C05:TempTemp2.cpp
// A multi-variate template template parameter.
#include <cstdint>
#include <iostream>
using namespace std;

template<class T, size_t N> class Array {
    T data[N];
    size_t count;
public:
    Array() { count = 0; }
    void push_back(const T& t) {
        if(count < N)
            data[count++] = t;
    }
    void pop_back() {
        if(count > 0)
            --count;
    }
    T* begin() { return data; }
}

```



```

    T* end() { return data + count; }
};

template<class T, size_t N, template<class, size_t> class Seq>
class Container {
    Seq<T, N> seq;
public:
    void append(const T& t) { seq.push_back(t); }
    T* begin() { return seq.begin(); }
    T* end() { return seq.end(); }
};

int main() {
    const size_t N = 10;
    Container<int, N, Array> container;
    container.append(1);
    container.append(2);
    int* p = container.begin();
    while(p != container.end())
        cout << *p++ << endl;
} ///~

```

再说明一次，在**Container**的声明内部，**Seq**的声明中参数名称不是必需的，但是需要有两个参数来声明数据成员**seq**，所以无类型参数**N**出现在模板型参数前面。

结合一下默认参数和模板型的模板参数就会发现一些细微的深一层问题。当编译器看到模板型模板参数的内部参数时，无法顾及到默认参数，因此为了得到一个确切的匹配，必须重复声明默认参数。下面的例子中，在固定大小的**Array**模板中使用了一个默认参数，这个例子也显示了如何在C++语言中适应这个古怪的举动。

```

//: C05:TempTemp3.cpp {-bor}{-msc}
// Template template parameters and default arguments.
#include <cstdint>
#include <iostream>
using namespace std;

template<class T, size_t N = 10> // A default argument
class Array {
    T data[N];
    size_t count;
public:
    Array() { count = 0; }
    void push_back(const T& t) {
        if(count < N)
            data[count++] = t;
    }
    void pop_back() {
        if(count > 0)
            --count;
    }
    T* begin() { return data; }
    T* end() { return data + count; }
};

template<class T, template<class, size_t = 10> class Seq>
class Container {
    Seq<T> seq; // Default used
public:
    void append(const T& t) { seq.push_back(t); }
    T* begin() { return seq.begin(); }
    T* end() { return seq.end(); }
};

```



```
int main() {
    Container<int, Array> container;
    container.append(1);
    container.append(2);
    int* p = container.begin();
    while(p != container.end())
        cout << *p++ << endl;
} ///:~
```

在下面这行语句中默认值的大小为10是必须的：

```
template<class T, template<class, size_t = 10> class Seq>
```

不管是在**Container**中**seq**的定义，还是在**main()**中**container**的定义都使用了默认值。本例与**TempTemp2.cpp**惟一的不同点，就是使用了默认值。这也是与前面所陈述的规则——即默认参数在一个编辑单元内仅能出现一次——惟一的例外。

由于标准序列容器（**vector**、**list**和**deque**，它们将在第7章中深入讨论）都有一个默认的分配器参数，上面讲解到的技术能帮助实现我们曾经有过的一个想法：传递这些序列容器中的一个作为模板参数。下面的程序分别传递**vector**模板类型参数和**list**模板类型参数创建了**Container**的两个实例：

```
///: C05:TempTemp4.cpp {-bor}{-msc}
// Passes standard sequences as template arguments.
#include <iostream>
#include <list>
#include <memory> // Declares allocator<T>
#include <vector>
using namespace std;

template<class T, template<class U, class = allocator<U> >
        class Seq>
class Container {
    Seq<T> seq; // Default of allocator<T> applied implicitly
public:
    void push_back(const T& t) { seq.push_back(t); }
    typename Seq<T>::iterator begin() { return seq.begin(); }
    typename Seq<T>::iterator end() { return seq.end(); }
};

int main() {
    // Use a vector
    Container<int, vector> vContainer;
    vContainer.push_back(1);
    vContainer.push_back(2);
    for(vector<int>::iterator p = vContainer.begin();
        p != vContainer.end(); ++p) {
        cout << *p << endl;
    }
    // Use a list
    Container<int, list> lContainer;
    lContainer.push_back(3);
    lContainer.push_back(4);
    for(list<int>::iterator p2 = lContainer.begin();
        p2 != lContainer.end(); ++p2) {
        cout << *p2 << endl;
    }
} ///:~
```

这里命名了内部模板**Seq**的第1个参数（使用名字**U**），这是因为标准序列容器的分配器必须使用与序列容器中所包含对象的类型相同的类型对自己进行参数化。同时，还由于默认的

**allocator**参数是已知的，就可以像在前述程序中一样，在随后引用的**Seq<T>**中省略掉它。然而，要想彻底地解释清楚这个例子，还必须讨论一下**typename**这个关键字的语义：

#### 5.1.4 typename关键字

考虑下面的程序：

```

//: C05:TypenamedID.cpp {-bor}
// Uses 'typename' as a prefix for nested types.

template<class T> class X {
    // Without typename, you should get an error:
    typename T::id i;
public:
    void f() { i.g(); }
};

class Y {
public:
    class id {
    public:
        void g() {}
    };
};

int main() {
    X<Y> xy;
    xy.f();
} ///:~

```

这个模板定义假定，处理的类**T**必须拥有某种称为**id**的嵌套标识符。**id**也可以是一个**T**的静态数据成员，这样就可以直接对**id**进行操作，但却不能“创建类型**id**”的“某个对象”，在这个例子中，标识符**id**被当作**T**内的一个嵌套类型处理。至于类**Y**，**id**本来就是它的一个嵌套类型（没有**typename**关键字），但编译器在编译类**X**的时候却根本不知道这些。

当模板中出现一个标识符时，若编译器可以在把这个标识符当作一个类型，或把它当作一个除类型之外的其他元素之间进行选择的话，则编译器将不会认为这个标识符是一个类型。也就是说，它会认为这个标识符指的是一个对象（其中包括那些基本类型的变量），或者是一个枚举，或者是其他什么。但是它绝不会——也不可能——认为它是一个类型。

由于在上述两种情况下，编译器默认的行为不会认为一个标识符名称是一个类型，因此必须对嵌套的名称使用**typename**关键字进行说明（除了在构造函数的初始化列表中，这时它的出现既不是必要的也不是允许的）。在上例中，当编译器看到**typename T::id**，它就会明白（由于关键字**typename**）**id**指的是一个嵌套类型，之后它就可以创建一个这个类型的对象了。

这个规则的简化叙述就是：若一个模板代码内部的某个类型被模板类型参数所限定，则必须使用关键字**typename**作为前缀进行声明，除非它已经出现在基类的规格说明中，或者它出现在同一作用域范围内的初始化列表中（这种情况下一定不要使用**typename**关键字）。

上面解释了关键字**typename**在程序**TempTemp4.cpp**中的使用。没有它，编译器就不会认为**Seq<T>::iterator**表达式是一个类型，而在程序中却要用它来定义成员函数**begin()**和**end()**的返回类型。

下面的例子定义了一个函数模板，它能够打印任意标准C++序列容器中的数据，这个例子使用了与**typename**类似的一种用法：



```

//: C05:PrintSeq.cpp {-msc}{-mwcc}
// A print function for Standard C++ sequences.
#include <iostream>
#include <list>
#include <memory>
#include <vector>
using namespace std;

template<class T, template<class U, class = allocator<U> >
        class Seq>
void printSeq(Seq<T>& seq) {
    for(typename Seq<T>::iterator b = seq.begin();
        b != seq.end();)
        cout << *b++ << endl;
}

int main() {
    // Process a vector
    vector<int> v;
    v.push_back(1);
    v.push_back(2);
    printSeq(v);
    // Process a list
    list<int> lst;
    lst.push_back(3);
    lst.push_back(4);
    printSeq(lst);
} ///:~

```

同前面一样，若没有**typename**关键字，编译器就会把**iterator**看作是**Seq<T>**的一个静态数据成员，这是一个语法错误，因为这里要求它是一个类型。

#### 1. 创建一个新类型

有一点很重要：一定不能认为关键字**typename**创建了一个新类型名。它确实没有。它的目的就是要通知编译器，被限定的那个标识符应该被解释为一个类型。请看下面这行语句：

```
typename Seq<T>::iterator It;
```

它产生一个名为**It**的变量，该变量被声明为**Seq<T>::iterator**类型。若想创建一个新类型名，通常应该使用关键字**typedef**，如下所示：

```
typedef typename Seq<It>::iterator It;
```

#### 2. 用**typename**代替**class**

关键字**typename**的另一个作用是，可以在模板定义的模板参数列表中选择使用**typename**代替**class**：

```

//: C05:UsingTypename.cpp
// Using 'typename' in the template argument list.

template<typename T> class X {};

int main() {
    X<int> x;
} ///:~

```

对大多数程序而言，这种描述方式使得代码更加一目了然。

### 5.1.5 以**template**关键字作为提示

当一个类型标识符不是预期的标识符时，正好**typename**关键字可以帮助编译器识别它们，

但编译器却还存在一些潜在的困难，比如‘<’字符和‘>’字符，它们不是标识符而是标记号(token)。有时它们代表小于号或大于号，而有时它们又作为模板参数列表的界定符。在这里，再次用**bitset**类作为例子来说明这个问题：

```
//: C05:DotTemplate.cpp
// Illustrate the .template construct.
#include <bitset>
#include <cstdint>
#include <iostream>
#include <string>
using namespace std;

template<class charT, size_t N>
basic_string<charT> bitsetToString(const bitset<N>& bs) {
    return bs. template to_string<charT, char_traits<charT>,
        allocator<charT> >();
}

int main() {
    bitset<10> bs;
    bs.set(1);
    bs.set(5);
    cout << bs << endl; // 0000100010
    string s = bitsetToString<char>(bs);
    cout << s << endl; // 0000100010
} ///:~
```

类**bitset**通过它的**to\_string**成员函数支持向字符串对象的转换。为了支持向多种字符串类的转换，**to\_string**本身就做成了一个模板，它是根据第3章讨论的**basic\_string**模板模式创建的。**bitset**中的**to\_string**的声明如下所示：

```
template<class charT, class traits, class Allocator>
basic_string<charT, traits, Allocator> to_string() const;
```

上面的**bitsetToString( )**函数模板可以要求用不同类型的字符串表示**bitset**。例如，若想获得一个宽字符串，可以改写成如下的调用形式：

```
wstring s = bitsetToString<wchar_t>(bs);
```

注意，**basic\_string**使用了默认的模板参数，这样在返回值中就不必重复**char\_traits**和**allocator**参数。遗憾的是，**bitset::to\_string**没有使用默认参数。使用**bitsetToString<char>(bs)**比每次都写出长长的完全地限制调用**bs.template to\_string<char, char\_traits, allocator<char> >()**要方便得多。

**bitsetToString( )**的返回语句中包含了**template**关键字，有趣的是，它位于作用在**bitset**对象**bs**上的点运算符之后的右边位置。使用这个关键字的原因是，如果解析这个模板，**to\_string**标记之后的“<”字符就会被解释为一个小于号而不是一个模板参数列表的开始标记。这里的**template**关键字的使用会告诉编译器，紧接着的是一个模板名称，这样“<”字符就会被正确地解释出来。基于同样的原因，这种用法也会用在运用于模板中的**->**和**::**的运算符上。与**typename**关键字一样，这种模板解析技术仅仅能用于模板代码<sup>①</sup>中。

#### 5.1.6 成员模板

**bitset::to\_string( )**函数模板是一个成员模板的例子：在另一个类或者类模板中声明的

① C++标准协会正在考虑解除这些解析提示仅仅适用于模板中的规则的限制，有一些编译器已经允许将它们用于非模板代码中。

一个模板。它允许一些独立的模板参数结合，以便组合使用。标准C++库中的**complex**类模板就是一个有用的例子。**complex**模板有一个类型参数，它代表一个拥有复数的实部和虚部的基浮点类型。下面的代码片段是从标准库中摘录出来的，它说明了在**complex**类模板中的成员模板构造函数：

```
template<typename T> class complex {
public:
    template<class X> complex(const complex<X>&);
```

标准的**complex**模板使用已有的类型如**float**、**double**和**long double**等对参数**T**进行特化。上面的成员模板构造函数创建了一个新复数，这个复数使用了另外一个浮点类型作为它的基类型，如下所示：

```
complex<float> z(1, 2);
complex<double> w(z);
```

在**w**的声明中，**complex**模板参数**T**是**double**类型，**X**是**float**类型。成员模板使得这种灵活的变换更加容易。

在模板中定义另一个模板是一种嵌套操作，如果想在外部类的定义之外定义成员模板，那么作为引入模板的前缀必须能够反映这种嵌套。例如，如果要想实现**complex**类的模板，还想在**complex**模板类定义之外定义成员模板构造函数，可以如下定义：

```
template<typename T>
template<typename X>
complex<T>::complex(const complex<X>& c) { /* Body here... */ }
```

标准库中成员函数模板的另一个应用是在容器的初始化中。假设有一个**int**型的**vector**，要想用它初始化一个新的**double**型的**vector**，如下所示：

```
int data[5] = { 1, 2, 3, 4, 5 };
vector<int> v1(data, data+5);
vector<double> v2(v1.begin(), v1.end());
```

只要**v1**中的元素与**v2**中的元素类型兼容（这里就是**double**型和**int**型）即可。**vector**类模板有如下成员模板构造函数：

```
template<class InputIterator>
vector(InputIterator first, InputIterator last,
       const Allocator& = Allocator());
```

这个构造函数在**vector**声明中使用了两次。**v1**用**int**型数组进行初始化时，**InputIterator**的类型是**int\***。**v2**使用**v1**进行初始化时，使用了成员模板构造函数的一个实例，用**InputIterator**表示**vector<int>::iterator**。

成员模板也可以是类（不一定必须是函数）。下面的例子说明了一个外部类模板内的成员类模板：

```
//: C05:MemberClass.cpp
// A member class template.
#include <iostream>
#include <typeinfo>
using namespace std;

template<class T> class Outer {
public:
    template<class R> class Inner {
    public:
        void f();
    };
};
```

```
};

template<class T> template<class R>
void Outer<T>::Inner<R>::f() {
    cout << "Outer == " << typeid(T).name() << endl;
    cout << "Inner == " << typeid(R).name() << endl;
    cout << "Full Inner == " << typeid(*this).name() << endl;
}

int main() {
    Outer<int>::Inner<bool> inner;
    inner.f();
} ///:~
```

在第8章中将会详细阐述**typeid**运算符，它只有一个参数并返回一个**type\_info**对象，这个对象的**name()**函数生成一个表示参数类型的字符串。例如，**typeid(int).name()**返回字符串“int”（实际的返回值与具体的操作平台有关）。**typeid**运算符也可以用一个表达式作参数，返回一个代表这个表达式类型的**type\_info**对象，例如，对于**int i**，**typeid(i).name()**返回的内容类似“int，”而**typeid(& i).name()**返回的内容类似“int\*”。

上述程序的输出应该如下所示：

```
Outer == int
Inner == bool
Full Inner == Outer<int>::Inner<bool>
```

主程序中变量**inner**的声明同时实例化了**Inner<bool>**和**Outer<int>**。

成员模板函数不能被声明为**virtual**类型。当今的编译器技术在解析一个类时，希望知道这个类的虚函数表的大小。如果允许虚成员模板函数的存在，则需要提前知道程序中所有这些成员函数的调用在什么位置。这是很不灵活的，尤其是在多文件项目中。

## 5.2 有关函数模板的几个问题

正如一个类模板描述了一族类，一个函数模板描述了一族函数。产生每种模板类型的语法本质上是相同的，只是在如何使用上有点区别。当在实例化类模板时总是需要使用尖括号并且提供所有的非默认模板参数。然而，对于函数模板，经常可以省略掉模板参数，甚至根本不允许使用默认模板参数。仔细看一下在**<algorithm>**头文件中声明的**min()**函数模板的实现，如下所示：

```
template<typename T> const T& min(const T& a, const T& b) {
    return (a < b) ? a : b;
}
```

可以通过提供尖括号里面的参数类型来调用这个模板，正如对类模板的操作一样，如下所示：

```
int z = min<int>(i, j);
```

这个语法告诉编译器，**min()**模板需要在参数**T**的位置使用**int**来进行特化，这样编译器就会产生相应的代码。依据从类模板中产生类的命名模式，可以认为这个实例化的函数名称是**min<int>()**。

### 5.2.1 函数模板参数的类型推断

可以像上面的例子一样，一直使用这样明确的函数模板特化方式，但是不明确指定模板参数类型，而让编译器从函数的参数中推断出它们的类型将会更方便，如下所示：

```
int z = min(i, j);
```

如果*i*和*j*都是`int`类型，编译器会知道程序需要的是`min<int>()`，之后它会自动进行实例化。由于在模板定义时指定了惟一的模板类型参数用于函数的两个参数，因此这两个参数的类型必须一致。对于由一个模板参数来限定类型的函数参数，C++系统不能提供标准转换。例如，若想在两个不同类型的参数（一个`int`型和一个`double`型）中找出其中的最小值，下面的这种`min()`调用将会出错：

```
int z = min(x, j); // x is a double
```

由于*x*和*j*是不同的类型，没有单一的参数与`min()`定义中的模板参数*T*匹配，因此这个调用与模板声明也不匹配。要解决这个问题，可以将一个参数的类型转换为另一个参数的类型，或者恢复到完全说明调用语法，如下所示：

```
int z = min<double>(x, j);
```

该语句告诉编译器产生`double`版本的`min()`，之后*j*通过标准类型转换规则向上转型成`double`型数据（因为函数`min<double>(const double&, const double&)`会自动产生转换）。

也可以要求`min()`的两个参数类型完全独立，如下所示：

```
template<typename T, typename U>
const T& min(const T& a, const U& b) {
    return (a < b) ? a : b;
}
```

这通常会是一个好办法，但由于`min()`必须返回一个值，却没有一个理想的方式来决定这个返回值的类型到底是*T*还是*U*，因此这里的这个“好办法”还是有问题的。

若一个函数模板的返回类型是一个独立的模板参数，当调用它的时候就一定要明确指定它的类型，因为这时已经无法从函数参数中推断出它的类型了。下面的`fromString`模板就是这样的例子。

```
//: C05:StringConv.h
// Function templates to convert to and from strings.
#ifdef STRINGCONV_H
#define STRINGCONV_H
#include <string>
#include <sstream>

template<typename T> T fromString(const std::string& s) {
    std::istringstream is(s);
    T t;
    is >> t;
    return t;
}

template<typename T> std::string toString(const T& t) {
    std::ostringstream s;
    s << t;
    return s.str();
}
#endif // STRINGCONV_H ///:~
```

这些函数模板提供了`std::string`与任意类型之间的转换，二者分别给出了一个流类插入符和提取符。下面是一个使用了包含在标准库中复数（`complex`）类的测试程序：

```
//: C05:StringConvTest.cpp
#include <complex>
#include <iostream>
#include "StringConv.h"
using namespace std;
```

```

int main() {
    int i = 1234;
    cout << "i == \" << toString(i) << \"\" << endl;
    float x = 567.89;
    cout << "x == \" << toString(x) << \"\" << endl;
    complex<float> c(1.0, 2.0);
    cout << "c == \" << toString(c) << \"\" << endl;
    cout << endl;

    i = fromString<int>(string("1234"));
    cout << "i == " << i << endl;
    x = fromString<float>(string("567.89"));
    cout << "x == " << x << endl;
    c = fromString<complex<float>>(string("(1.0,2.0)"));
    cout << "c == " << c << endl;
} ///:~

```

输出和预期的结果相同：

```

i == "1234"
x == "567.89"
c == "(1,2)"

i == 1234
x == 567.89
c == (1,2)

```

注意，在每一个**fromString()**的实例化调用中，都指定了模板参数。如果有一个函数模板，它的模板参数既作为参数类型又作为返回类型，那么一定要首先声明函数的返回类型参数，否则就不能省略掉函数参数表中的任何类型参数。作为一个示例，看看下面这个著名的函数模板：<sup>①</sup>

```

//: C05:ImplicitCast.cpp

template<typename R, typename P>
R implicit_cast(const P& p) {
    return p;
}

int main() {
    int i = 1;
    float x = implicit_cast<float>(i);
    int j = implicit_cast<int>(x);
    //! char* p = implicit_cast<char*>(i);
} ///:~

```

如果将程序中靠近文件顶部的模板参数列表中的**R**和**P**交换一下，这个程序将不能通过编译，这是因为没有指定函数的返回类型（第1个模板参数将作为函数的参数类型）。最后一行（被注解掉的）也是不合法的用法，原因是没有从**int**到**char\***的标准类型转换。**implicit\_cast**显示了代码中允许的类型转换。

稍加注意，甚至可以用这种办法推断出数组的维数。下面的例子中有一个数组初始化函数模板（**init2**），它进行了这样的推断：

```

//: C05:ArraySize.cpp
#include <cstddef>
using std::size_t;

template<size_t R, size_t C, typename T>

```

① 参见Stroustrup, 《The C++ Programming Language》，第3版，Addison Wesley，第335~336页。

```

void init1(T a[R][C]) {
    for(size_t i = 0; i < R; ++i)
        for(size_t j = 0; j < C; ++j)
            a[i][j] = T();
}

template<size_t R, size_t C, class T>
void init2(T (&a)[R][C]) { // Reference parameter
    for(size_t i = 0; i < R; ++i)
        for(size_t j = 0; j < C; ++j)
            a[i][j] = T();
}

int main() {
    int a[10][20];
    init1<10,20>(a); // Must specify
    init2(a);         // Sizes deduced
} ///:~

```

数组维数没有被作为函数参数类型的一部分进行传递，除非这个参数是指针或引用。函数模板**init2**声明了**a**是一个二维数组的引用，因此它的维数**R**和**C**可由模板很容易地推断出来，这样就使得**init2**可以非常方便地初始化一个任意大小的二维数组。模板**init1**不是通过引用来传递数组，因此数组的大小必须被明确指定，尽管也可以推断出它的类型参数。

### 5.2.2 函数模板重载

像普通函数一样，也可以用相同的函数名重载函数模板。编译器在处理程序中的函数调用时，它必须能够知道哪一个模板或普通函数是最适合调用的函数。

接着前面介绍的**min()**函数模板，在这里再添加几个普通函数：

```

//: C05:MinTest.cpp
#include <cstring>
#include <iostream>
using std::strcmp;
using std::cout;
using std::endl;

template<typename T> const T& min(const T& a, const T& b) {
    return (a < b) ? a : b;
}

const char* min(const char* a, const char* b) {
    return (strcmp(a, b) < 0) ? a : b;
}

double min(double x, double y) {
    return (x < y) ? x : y;
}

int main() {
    const char *s2 = "say \"Ni-!\"", *s1 = "knights who";
    cout << min(1, 2) << endl; // 1: 1 (template)
    cout << min(1.0, 2.0) << endl; // 2: 1 (double)
    cout << min(1, 2.0) << endl; // 3: 1 (double)
    cout << min(s1, s2) << endl; // 4: knights who (const
                                // char*)
    cout << min<>(s1, s2) << endl; // 5: say "Ni-!"
                                // (template)
} ///:~

```



除了函数模板，这个程序还定义了两个非模板函数：一个C语言风格的字符串版本和一个**double**版本的**min()**函数。若这个程序中不存在函数模板，上面主函数第1行的函数调用将会调用**double**版本的**min()**函数，这是由于**int**型可以经标准转换为**double**型。由于模板能够产生一个**int**版本的**min()**函数，这肯定是最佳的匹配，因此事实上就是这样进行的。第2行中的调用是一个**double**版本的**min()**函数的准确匹配，第3行也调用了同一个函数，只是在内部将1转变成1.0。第4行中，直接调用了**min()**函数的**const char\***版本。第5行在函数名后加一对空的尖括号来强迫编译器使用模板，因此编译器从模板中生成它的一个**const char\***版本来使用（从它的错误输出可以证实——这个函数比较了两个字符串的地址！<sup>①</sup>）。如果想知道为什么在应该用**using namespace std**的地方使用了几个**using**声明，这是因为有些编译器在其中包含了**std::min()**的头文件，这将会与在程序中命名的**min()**声明发生冲突。

如上所述，只要编译器能够区分开，就可以重载同名的模板。例如可以声明一个包含3个参数的**min()**函数模板：

```
template<typename T>
const T& min(const T& a, const T& b, const T& c);
```

这个模板版本仅仅是为了调用带有3个同类型参数的**min()**函数而生成的。

### 5.2.3 以一个已生成的函数模板地址作为参数

在很多情况下需要获得一个函数的地址。例如，可以生成一个函数，它的参数是一个指向另一个函数的指针。此处的另一个函数有可能就是由一个函数模板生成的，因此需要以某种方式来处理这种以函数模板的地址做参数的情况：<sup>②</sup>

```
///C05:TemplateFunctionAddress.cpp {-mwcc}
// Taking the address of a function generated
// from a template.

template<typename T> void f(T*) {}

void h(void (*pf)(int*)) {}

template<typename T> void g(void (*pf)(T*)) {}

int main() {
    h(&f<int>); // Full type specification
    h(&f); // Type deduction
    g<int>(&f<int>); // Full type specification
    g(&f<int>); // Type deduction
    g<int>(&f); // Partial (but sufficient) specification
} ///:~
```

这个例子说明了如下几个问题。首先，既然使用模板，所有的标识就必须匹配。函数**h()**有一个指针参数，这个指针指向一个函数——它有一个**int\***型参数，返回值类型为**void**。这个函数就是模板**f()**生成的函数。其次，拥有一个函数指针作参数的函数本身可以是一个模板，如本例中的函数模板**g()**。

也可以在**main()**中看到类型推断。第1个对**h()**的调用明确地给出了**f()**的模板参数，但

- 
- ① 从技术上说，对不在同一个数组中的两个指针的比较是一种不明确的行为，但如今的编译器不再对此做出解释。所有要这样做的理由都认为是正确的。
  - ② 感谢Nathan Myers提供了这个例子。



由于**h()**规定只接收具有**int\***参数的函数地址作参数，因此第2个调用由编译器来推断类型。至于**g()**，它的情况就更加有趣了，因为它在其中引用了两个模板。如果什么都不给，编译器就推断不出类型；但若说明了一个**int**，或者赋予**f()**或者赋予**g()**，余下的类型编译器自己就能够推断出来。

当想把在**<cctype>**中声明的**tolower**或**toupper**传递给函数做参数时，就会出现一个模糊的问题。例如，在编程中，有可能使用它们和**transform**算法（将在下一章详细阐述）将一个字符串转变成小写或者大写。必须小心使用，因为存在多个有关这些函数的声明。一个初学者的使用方法可能会像下面这样：

```
// The variable s is a std::string
transform(s.begin(), s.end(), s.begin(), tolower);
```

**transform**算法的第4个参数（在这个例子中就是**tolower()**）作用到字符串**s**中的每一个字符上，这个算法还把结果写回**s**中，也就是将**s**中的每一个字符都用它的小写形式进行重写。作为一条语句把它写在那里，但这个语句有可能执行，也可能根本就没有执行！在下面的情况下它就执行失败了：

```
//: C05:FailedTransform.cpp {-xo}
#include <algorithm>
#include <cctype>
#include <iostream>
#include <string>
using namespace std;
int main() {
    string s("LOWER");
    transform(s.begin(), s.end(), s.begin(), tolower);
    cout << s << endl;
} ///:~
```

即使编译器让这个程序侥幸通过，它也是不合法的。原因是**<iostream>**头文件中也建造了可利用的具有两个参数的**tolower()**和**toupper()**版本：

```
template<class charT> charT toupper(charT c,
                                   const locale& loc);
template<class charT> charT tolower(charT c,
                                   const locale& loc);
```

这两个函数模板的第2个参数是**locale**类型参数。在上面的程序中，编译器无法得知它应该使用**<cctype>**中定义的具有一个参数的**tolower()**版本还是上述的版本。可以（几乎可以！）用**transform**调用中的类型转换来解决这个问题，如下所示：

```
transform(s.begin(), s.end(), s.begin(),
         static_cast<int (*)(>int)>(tolower));
```

（用**int**代替**char**后重新调用**tolower()**和**toupper()**函数执行。）上面的类型转换很清楚地表达了想要使用具有一个参数的**tolower()**函数版本的期望。这种做法对某些编译器可能会成功，但并不是所有的编译器都是如此。其原因有点晦涩难懂：在C语言中允许一个库的实现连接“C连接”（意味着函数名称不包含所有的辅助信息<sup>①</sup>，而正常的C++函数却包含）到从C语言继承过来的函数。如果是这样的话，类型转换则失败：因为**transform**是一个C++函数模板，它期待它的第4个参数进行C++连接——并且类型转换也不允许改变这种连接。我们又陷入了困境！

① 例如在一个修饰的名称中被编码的类型信息。

解决的办法是在一个语义明确的语境中调用**tolower()**。例如，可以编写一个名叫**strTolower()**的函数，将它放在一个不包含**<iostream>**的独立的文件中，如下所示：

```
///C05:StrTolower.cpp {0} {-mwcc}
#include <algorithm>
#include <cctype>
#include <string>
using namespace std;

string strTolower(string s) {
    transform(s.begin(), s.end(), s.begin(), tolower);
    return s;
} ///:~
```

该程序没有包含头文件**<iostream>**，在这种语境中，程序使用的编译器就不会引入带有两个参数的**tolower()**版本<sup>①</sup>，当然也就不会产生任何问题。经过这样处理之后，就可以正常使用这个函数了：

```
///C05:Tolower.cpp {-mwcc}
///{L} StrTolower
#include <algorithm>
#include <cctype>
#include <iostream>
#include <string>
using namespace std;
string strTolower(string);

int main() {
    string s("LOWER");
    cout << strTolower(s) << endl;
} ///:~
```

另一个解决办法是写一个经过封装的函数模板，用来清楚地调用正确的**tolower()**版本：

```
///C05:ToLower2.cpp {-mwcc}
#include <algorithm>
#include <cctype>
#include <iostream>
#include <string>
using namespace std;

template<class charT> charT strTolower(charT c) {
    return tolower(c); // One-arg version called
}

int main() {
    string s("LOWER");
    transform(s.begin(), s.end(), s.begin(), &strTolower<char>);
    cout << s << endl;
} ///:~
```

这种版本有一个好处，即由于基础字符类型是一个模板参数，因而该模板既可以处理宽字符串，也可以处理窄字符串。C++标准委员会正致力于修订语言，使得第1个例子（没有使用类型转换的）能够执行，也许过不了多久这些外围工作就可以被忽略了（由C++标准来完成）。<sup>②</sup>

① C++编译器能够引入它们想要的任何地方的名称，然而幸运的是，大多数编译器对自己不需要的名称不会进行声明。

② 若对关于C++语言修改的建议感兴趣，可以参看Core Issue 352。

### 5.2.4 将函数应用到STL序列容器中

假设要想获得一个STL序列容器（更多的内容将在后面的章节中学习，现在只用到STL序列容器家族中的**vector**），并且想将一个成员函数应用到这个容器包含的所有对象中。因为一个**vector**可以包含任意类型的对象，这就需要一个可以应用到任意类型的**vector**对象的函数：

```

//: C05:ApplySequence.h
// Apply a function to an STL sequence container.

// const, 0 arguments, any type of return value:
template<class Seq, class T, class R>
void apply(Seq& sq, R (T::*f)() const) {
    typename Seq::iterator it = sq.begin();
    while(it != sq.end())
        ((*it++)->*f)();
}

// const, 1 argument, any type of return value:
template<class Seq, class T, class R, class A>
void apply(Seq& sq, R (T::*f)(A) const, A a) {
    typename Seq::iterator it = sq.begin();
    while(it != sq.end())
        ((*it++)->*f)(a);
}

// const, 2 arguments, any type of return value:
template<class Seq, class T, class R,
        class A1, class A2>
void apply(Seq& sq, R (T::*f)(A1, A2) const,
        A1 a1, A2 a2) {
    typename Seq::iterator it = sq.begin();
    while(it != sq.end())
        ((*it++)->*f)(a1, a2);
}

// Non-const, 0 arguments, any type of return value:
template<class Seq, class T, class R>
void apply(Seq& sq, R (T::*f)()) {
    typename Seq::iterator it = sq.begin();
    while(it != sq.end())
        ((*it++)->*f)();
}

// Non-const, 1 argument, any type of return value:
template<class Seq, class T, class R, class A>
void apply(Seq& sq, R (T::*f)(A), A a) {
    typename Seq::iterator it = sq.begin();
    while(it != sq.end())
        ((*it++)->*f)(a);
}

// Non-const, 2 arguments, any type of return value:
template<class Seq, class T, class R,
        class A1, class A2>
void apply(Seq& sq, R (T::*f)(A1, A2),
        A1 a1, A2 a2) {
    typename Seq::iterator it = sq.begin();
    while(it != sq.end())
        ((*it++)->*f)(a1, a2);
}
// Etc., to handle maximum likely arguments ///:~

```



上面的**apply()**函数模板有一个对容器类进行引用的引用参数，还有一个指针参数，它指向容器类中包含的对象的一个成员函数。这个模板使用一个迭代器遍历该序列，并且在每个对象上调用这个成员函数。现在已经重载了**const**版本的函数模板，这样一来，在**const**和非**const**函数中就都可以调用这个成员函数了。

注意，在**applySequence.h**中没有包含STL头文件（也没有包含其他的头文件），因此它并不局限于只能用于STL容器。然而，它的假设（主要是由于**iterator**的名称及行为）确实是用于STL序列容器中，而且它也假设容器的元素都是指针类型。

读者可以看到有多个**apply()**版本，更进一步地说明了函数模板的重载。尽管这些模板允许返回任意类型的值（这点被忽略了，但类型信息要求匹配指向成员函数的指针），每一个版本都带有不同个数的参数，并且由于这是模板，因此它们的参数可以是任意类型。在此惟一的缺陷，就是没有提供一个可以产生模板的“超模板”；读者必须亲自决定究竟需要几个参数来选用合适的模板定义。

为了测试一下**apply()**的这几个重载版本，这里创建了一个类**Gromit**<sup>①</sup>，它包含几个带有不同个数参数的函数，以及由**const**和非**const**的成员函数：

```
//: C05:Gromit.h
// The techno-dog. Has member functions
// with various numbers of arguments.
#include <iostream>

class Gromit {
    int arf;
    int totalBarks;
public:
    Gromit(int arf = 1) : arf(arf + 1), totalBarks(0) {}
    void speak(int) {
        for(int i = 0; i < arf; i++) {
            std::cout << "arf! ";
            ++totalBarks;
        }
        std::cout << std::endl;
    }
    char eat(float) const {
        std::cout << "chomp!" << std::endl;
        return 'z';
    }
    int sleep(char, double) const {
        std::cout << "zzz..." << std::endl;
        return 0;
    }
    void sit() const {
        std::cout << "Sitting..." << std::endl;
    }
}; ///:~
```

现在，可以用**apply()**模板函数来调用**vector<Gromit\*>**对象中包含的**Gromit**的成员函数了，如下所示：

```
//: C05:ApplyGromit.cpp
// Test ApplySequence.h.
#include <cstdint>
#include <iostream>
```

① 参考由Nick Park 出品的描写Wallace和Gromit英国动画短片。

```

#include <vector>
#include "ApplySequence.h"
#include "Gromit.h"
#include "../purge.h"
using namespace std;

int main() {
    vector<Gromit*> dogs;
    for(size_t i = 0; i < 5; i++)
        dogs.push_back(new Gromit(i));
    apply(dogs, &Gromit::speak, 1);
    apply(dogs, &Gromit::eat, 2.0f);
    apply(dogs, &Gromit::sleep, 'z', 3.0);
    apply(dogs, &Gromit::sit);
    purge(dogs);
} ///:~

```

**purge()** 函数是一个小型的实用程序，该函数调用 **delete** 来清除序列上的所有元素。读者将会在第7章找到它的定义，并且本教材在许多的地方都会用到它。

尽管 **apply()** 的定义有点复杂，而且不像读者所希望的那样使初学者都可以理解，但是它使用起来却非常简单明了，初学者也可以在使用中知道它企图完成什么功能，而不必知道它是如何完成的。这也是所有程序组件应该追求的目标。复杂的细节由程序组件的设计者来完成。而用户只关心完成他们的目标，他们不必看、不必了解、也不依赖那些底层实现的细节。在下一章中要探索将函数应用到序列容器中的更灵活的方式。

### 5.2.5 函数模板的半有序

前面曾经提到过，使用像 **min()** 函数这样的普通函数重载，比使用函数模板更可取。如果一个函数可以匹配某个函数调用，为什么还要再生成另外一个函数呢？在缺少普通函数时，对函数模板进行重载有可能引起二义性 (ambiguity) 的情况，即不知选择哪个模板。为了将发生这种情况的几率降到最低，系统为这些函数模板定义了次序 (ordering)，在生成模板函数的时候，编译器将从这些函数模板中选择特化程度最高 (most specialized) 的模板 (如果有这种模板的话)。一个函数模板要考虑多种特化，在这些特化的模板中对于某个特定的函数模板来说，如果每一种可能的参数列表的选择都能够匹配该模板的参数列表，那么，这些可能的参数列表选择也都能够匹配另一个函数模板的参数列表，但反过来却不成立。请看下面的函数模板声明，取自C++标准文档：

```

template<class T> void f(T);
template<class T> void f(T*);
template<class T> void f(const T*);

```

任何类型都可以匹配第1个模板。第2个模板比第1个模板的特化程度更高，因为只有指针类型才能够匹配它。换句话说，可以把匹配第2个模板的一组可能的函数调用看做匹配第1个模板的子集。上面的第2个模板和第3个模板的声明也存在类似的关系：第3个仅仅能被指向 **const** 的指针匹配调用，但第2个模板包含了任意的指针类型。下面的程序说明了这些规则：

```

//: C05:PartialOrder.cpp
// Reveals ordering of function templates.
#include <iostream>
using namespace std;

template<class T> void f(T) {
    cout << "T" << endl;
}

```

```

    template<class T> void f(T*) {
        cout << "T*" << endl;
    }
    template<class T> void f(const T*) {
        cout << "const T*" << endl;
    }

    int main() {
        f(0);           // T
        int i = 0;
        f(&i);          // T*
        const int j = 0;
        f(&j);          // const T*
    } //::~~

```

**f(&i)**调用和第1个模板匹配，但由于第2个模板的特化程度更高，因此这里调用了第2个模板。在此处第3个模板不能被调用，这是因为该指针不是指向**const**的指针。**f(&j)**调用匹配了所有这3个模板（比如，在第2个模板中的**T**就是**const int**），但是由于同样的原因，第3个模板特化程度更高，因此实际上调用了它。

如果在一组重载的函数模板中没有“特化程度最高”的模板，则会出现二义性，编译器将会报错。这就是为什么把这种特征叫做“半有序（partial ordering）”的缘故——它不可能完全解决所有可能出现的情况。类似的规则同样也存在于类模板中（参见5.3.2节）。

### 5.3 模板特化

术语特化（specialization）在C++中有一个特别的与模板相关的含义。从本质上说，一个模板定义就是一个实体一般化（generalization）的过程，因为它在一般条件下描述了某个范围内的一族函数或类。给定模板参数时，这些模板参数决定了这一族函数或类的许多可能的实例中的一个独一无二的实例，因此这样的结果就被称做模板的一个特化。本章开始时介绍的**min()**函数模板是一个寻找最小值函数的一般化，因为没有指定它的参数类型。若为这个模板参数提供了类型，不管它是明确给定的还是通过参数推断获得的，由编译器生成的结果代码（例如，**min <int>()**）都是这个模板的一个特化。生成的代码也被认为是这个模板的一个实例化（instantiation），就像是由模板工具完全生成它的整个代码体一样。

#### 5.3.1 显式特化

编程人员也可以自己为一个模板提供代码来使其特化，采用这种方法进行编码的程序设计人员越来越多。类模板经常需要程序员为它提供模板特化，在本节，将从**min()**函数模板开始介绍这个语法。

回忆一下本章前面讨论过的**MinTest.cpp**中下面的这个普通函数：

```

const char* min(const char* a, const char* b) {
    return (strcmp(a, b) < 0) ? a : b;
}

```

这是一个比较字符串而不是比较地址的**min()**调用。尽管这个例子在这里没有什么用处，但可以作为替代为**min()**定义一个**const char\***的特化，如下面的程序所示：

```

//: C05:MinTest2.cpp
#include <cstring>
#include <iostream>
using std::strcmp;
using std::cout;

```

```

using std::endl;

template<class T> const T& min(const T& a, const T& b) {
    return (a < b) ? a : b;
}

// An explicit specialization of the min template
template<>
const char* const& min<const char*>(const char* const& a,
                                     const char* const& b) {
    return (strcmp(a, b) < 0) ? a : b;
}

int main() {
    const char *s2 = "say \Ni-!\\"", *s1 = "knights who";
    cout << min(s1, s2) << endl;
    cout << min<>(s1, s2) << endl;
} ///:~

```

前缀“**template< >**”告诉编译器接下来的是一个模板的特化。模板特化的类型必须出现在函数名后紧跟的尖括号中，就像通常在一个明确指定参数类型的函数调用中一样。注意，程序在这个显式特化（explicit specialization）中仔细地（carefully）用**const char\***代替了**T**。一旦在最初的模板定义时使用了**const T**，则这个**const**就会修正所有的**T**类型。这里的**const**常量是一个指向**const char\***的指针。因此在模板特化时必须用**const char\* const**代替**const T**。当编译器在程序中看到一个带有**const char\***参数的**min()**调用时，它就会去实例化**min()**的**const char\***版本，这样该版本就可以被调用了。这个程序中的两次**min()**调用都是调用同一个**min()**的特化版本。

类模板显式特化往往比函数模板显式特化更有用。当程序员为一个类模板提供了一份完整的特化时，依然需要实现其中的所有成员函数。这是由于所提供的是一个单独的类，客户代码常常希望能提供完整的接口实现。

标准库中有一个**vector**的显式特化，该特化在它持有**bool**类型的对象时使用。**Vector <bool>**实现的功能是让库将位（bit）封装成整数（integer）来节省存储空间。<sup>①</sup>

如前所述，基本**vector**类模板的声明如下：

```

template<class T, class Allocator = allocator<T> >
class vector {...};

```

要为**bool**类型的对象进行特化，如下显式声明该特化：

```

template<> class vector<bool, allocator<bool> > {...};

```

这再一次很快地被认为是一个完整的显式特化，因为有**template< >**前缀，还因为所有基本的模板参数都由令人感到满意的一个参数列表附加到类名中。

结果证明，事实上**vector <bool>**比前面所描述的模板特化方法具有更好的灵活性，具体内容请参看下节。

### 5.3.2 半特化

类模板也可以半特化（partial specialization），这意味着在模板特化的某些方法中至少还有一个方法，其模板参数是“开放的”。**Vector <bool>**限定了对象类型（**bool**类型），但并没有指定参数**allocator**的类型。下面是一个实际的**vector <bool>**声明：

① 第7章将会详细讨论**vector<bool>**。

```
template<class Allocator> class vector<bool, Allocator>;
```

读者可以把它理解成半特化，因为在**template**关键字后面（还没有被指定的参数）和**class**关键字后面（已经指定了参数）的尖括号里都是非空的参数列表。由于**vector <bool>**以这种方式定义，用户就可以提供一个自定义的**allocator**类型，即使参数列表中包含的类型**bool**是不变的。换句话说，类模板的特化和这种特别的半特化，构成了一种“重载”的类模板。

#### 类模板的半有序

选用哪个类模板来进行实例化的规则类似于函数模板的半有序规则——应该选择“特化程度最高”的模板。在下面的程序中，各个**f()**成员函数里面的字符串解释了每个模板定义的职责：

```
//: C05:PartialOrder2.cpp
// Reveals partial ordering of class templates.
#include <iostream>
using namespace std;

template<class T, class U> class C {
public:
    void f() { cout << "Primary Template\n"; }
};

template<class U> class C<int, U> {
public:
    void f() { cout << "T == int\n"; }
};

template<class T> class C<T, double> {
public:
    void f() { cout << "U == double\n"; }
};

template<class T, class U> class C<T*, U> {
public:
    void f() { cout << "T* used\n"; }
};

template<class T, class U> class C<T, U*> {
public:
    void f() { cout << "U* used\n"; }
};

template<class T, class U> class C<T*, U*> {
public:
    void f() { cout << "T* and U* used\n"; }
};

template<class T> class C<T, T> {
public:
    void f() { cout << "T == U\n"; }
};

int main() {
    C<float, int>().f();    // 1: Primary template
    C<int, float>().f();    // 2: T == int
    C<float, double>().f(); // 3: U == double
    C<float, float>().f();  // 4: T == U
    C<float*, float>().f(); // 5: T* used [T is float]
    C<float, float*>().f(); // 6: U* used [U is float]
    C<float*, int*>().f();  // 7: T* and U* used [float,int]
    // The following are ambiguous:
    // 8: C<int, int>().f();
    // 9: C<double, double>().f();
```





```
// 10: C<float*, float*>().f();
// 11: C<int, int*>().f();
// 12: C<int*, int*>().f();
} ///:~
```

如同读者看到的那样，可以根据模板参数是否是指针类型，或者它们是否和特化参数类型相同来部分地指定模板参数。当用**T\***作为模板参数来进行模板特化时，如上例的主程序中的第5行，**T**本身不是最高级的被传递的指针类型——它是一个指针指向的类型（本例中是**float**）。**T\***特化是一种允许用指针类型来匹配的模式。如果用**int\*\***作为第1个模板参数，**T**就是**int\***。主程序中的第8行具有二义性，因为程序中既有把**int**作为第1个参数的模板，也有带有两个类型相同参数的独立模板——在这两个模板中无法进一步进行取舍。同样的逻辑错误存在于第9行到第12行。

### 5.3.3 一个实例

可以很容易地从一个类模板中派生出一个新的模板，也可以通过继承和实例化一个现存的模板来创建一个新的模板。例如，若**vector**模板已经完成了程序员想要做的绝大多数事情，但在某种应用中，还希望有一个能够自己进行排序的版本，则可以很容易地复用**vector**代码。下面的例子是建立一个**vector < T >**的派生类，而且添加了排序（sorting）功能。注意，该类派生自**vector**，由于其没有虚析构函数，当需要在析构函数中执行清除操作的时候，这个派生类就会很危险。

```
//: C05:Sortable.h
// Template specialization.
#ifndef SORTABLE_H
#define SORTABLE_H
#include <cstring>
#include <cstdint>
#include <string>
#include <vector>
using std::size_t;

template<class T>
class Sortable : public std::vector<T> {
public:
    void sort();
};

template<class T>
void Sortable<T>::sort() { // A simple sort
    for(size_t i = this->size(); i > 0; --i)
        for(size_t j = 1; j < i; ++j)
            if(this->at(j-1) > this->at(j)) {
                T t = this->at(j-1);
                this->at(j-1) = this->at(j);
                this->at(j) = t;
            }
}

// Partial specialization for pointers:
template<class T>
class Sortable<T*> : public std::vector<T*> {
public:
    void sort();
};

template<class T>
void Sortable<T*>::sort() {
    for(size_t i = this->size(); i > 0; --i)
```



```

        for(size_t j = 1; j < i; ++j)
            if(*this->at(j-1) > *this->at(j)) {
                T* t = this->at(j-1);
                this->at(j-1) = this->at(j);
                this->at(j) = t;
            }
    }

    // Full specialization for char*
    // (Made inline here for convenience -- normally you would
    // place the function body in a separate file and only
    // leave the declaration here).
    template<> inline void Sortable<char*>::sort() {
        for(size_t i = this->size(); i > 0; --i)
            for(size_t j = 1; j < i; ++j)
                if(std::strcmp(this->at(j-1), this->at(j)) > 0) {
                    char* t = this->at(j-1);
                    this->at(j-1) = this->at(j);
                    this->at(j) = t;
                }
    }
}
#endif // SORTABLE_H ///:~

```

除了实例化类这个功能之外，**Sortable**模板对所有其他功能都有一个限制：它们必须包含一个>运算符。它只可以正确地处理非指针对象（包括系统内在类型的对象）。完全特化使用**strcmp()**对元素进行比较，并根据以空结束符来界定字符串的规则来对**char\***型的**vector**进行排序。上例中的“**this->**”是一个强制用法<sup>①</sup>，它将会在本章后面的“名字查找问题”一节中介绍<sup>②</sup>。

这里有一个**Sortable.h**的驱动程序，它使用了本章前面介绍过的随机数生成器：

```

//: C05:Sortable.cpp
//{-bor} (Because of bitset in Urand.h)
// Testing template specialization.
#include <cstdint>
#include <iostream>
#include "Sortable.h"
#include "Urand.h"
using namespace std;

#define asz(a) (sizeof a / sizeof a[0])

char* words[] = { "is", "running", "big", "dog", "a", };
char* words2[] = { "this", "that", "theother", };

int main() {
    Sortable<int> is;
    Urand<47> rnd;
    for(size_t i = 0; i < 15; ++i)
        is.push_back(rnd());
    for(size_t i = 0; i < is.size(); ++i)
        cout << is[i] << ' ';
    cout << endl;
    is.sort();
    for(size_t i = 0; i < is.size(); ++i)
        cout << is[i] << ' ';
}

```

① 可以使用任何其他合法的限定用法来代替**this->**，比如**Sortable::at()**或者**vector<T>::at()**。主要是它必须被限定。

② 也可参看第7章中**PriorityQueue6.cpp**的解释。

```

cout << endl;

// Uses the template partial specialization:
Sortable<string*> ss;
for(size_t i = 0; i < asz(words); ++i)
    ss.push_back(new string(words[i]));
for(size_t i = 0; i < ss.size(); ++i)
    cout << *ss[i] << ' ';
cout << endl;
ss.sort();
for(size_t i = 0; i < ss.size(); ++i) {
    cout << *ss[i] << ' ';
    delete ss[i];
}
cout << endl;

// Uses the full char* specialization:
Sortable<char*> scp;
for(size_t i = 0; i < asz(words2); ++i)
    scp.push_back(words2[i]);
for(size_t i = 0; i < scp.size(); ++i)
    cout << scp[i] << ' ';
cout << endl;
scp.sort();
for(size_t i = 0; i < scp.size(); ++i)
    cout << scp[i] << ' ';
cout << endl;
} ///:~

```

上面的每一个模板的实例化都使用了该模板的不同版本。**Sortable <int>** 使用的是基本的模板。**Sortable <string\*>** 使用了指针类型的半特化版本。最后，**Sortable <char\*>** 使用的是**char\***类型的完全特化模板。若没有这个完全特化版本，读者也许会误认为一切还会照原样正确无误地执行，因为**words**数组仍能排序出“a big dog is running”，这是由于半特化版本也可以完成每个数组的第1个字符的比较。然而，对于**words2**数组却不能够正确地排序。

### 5.3.4 防止模板代码膨胀

无论何时，一旦对某个类模板进行了实例化，伴随着所有在程序中调用的该模板的成员函数，类定义中用于对其进行详尽描述的特化代码也就会生成。只有被调用的成员函数才生成代码。这是不错的，读者可以在下面的程序中看到这一点：

```

//: C05:DelayedInstantiation.cpp
// Member functions of class templates are not
// instantiated until they're needed.

class X {
public:
    void f() {}
};

class Y {
public:
    void g() {}
};

template<typename T> class Z {
    T t;
public:
    void a() { t.f(); }
    void b() { t.g(); }
};

```



```
};

int main() {
    Z<X> zx;
    zx.a(); // Doesn't create Z<X>::b()
    Z<Y> zy;
    zy.b(); // Doesn't create Z<Y>::a()
} ///:~
```

在这里，尽管模板**Z**打算使用**T**的两个成员函数**f()**和**g()**，但实际上，当在程序中明确地为**zx**调用**Z<X>::a()**时候，程序在编译时就只能够生成**Z<X>::a()**的代码。（若同时也想生成**Z<X>::b()**的代码，则会产生一个编译时错误信息，因为它试图调用一个并不存在的**X::g()**。）同样的道理，对**zy.b()**的调用也不会生成**Z<Y>::a()**。结果是，**Z**模板可以跟类**X**和类**Y**在一起使用，但是，当类在进行第1次实例化时，如果所有的成员函数都生成了，就会使许多模板的使用明显地受到限制。

假设有一个模板化的**Stack**容器，现在想用**int**、**int\***和**char\***对它进行特化。这样将会生成3个版本的**Stack**代码，并链接为程序的一部分。使用模板的原因之一，首先就是不必手工复制代码；但是代码仍然被复制了——只不过是编译器代替程序员完成了这个工作而已。可以结合使用完全特化和半特化模板，将指针类型存储到某个独立的类中，这样可以减少程序实现的体积。其关键是用**void\***进行完全特化，然后从**void\***实现中派生出所有其他的指针类型，这样共同的代码就可以共享了。下面的程序说明了这个技术：

```
///: C05:Nobloat.h
// Shares code for storing pointers in a Stack.
#ifndef NOBLOAT_H
#define NOBLOAT_H
#include <cassert>
#include <cstddef>
#include <cstring>

// The primary template
template<class T> class Stack {
    T* data;
    std::size_t count;
    std::size_t capacity;
    enum { INIT = 5 };
public:
    Stack() {
        count = 0;
        capacity = INIT;
        data = new T[INIT];
    }
    void push(const T& t) {
        if(count == capacity) {
            // Grow array store
            std::size_t newCapacity = 2 * capacity;
            T* newData = new T[newCapacity];
            for(size_t i = 0; i < count; ++i)
                newData[i] = data[i];
            delete [] data;
            data = newData;
            capacity = newCapacity;
        }
        assert(count < capacity);
        data[count++] = t;
    }
    void pop() {
        assert(count > 0);
    }
};
```



```

        --count;
    }
    T top() const {
        assert(count > 0);
        return data[count-1];
    }
    std::size_t size() const { return count; }
};

// Full specialization for void*
template<> class Stack<void*> {
    void** data;
    std::size_t count;
    std::size_t capacity;
    enum { INIT = 5 };
public:
    Stack() {
        count = 0;
        capacity = INIT;
        data = new void*[INIT];
    }
    void push(void* const & t) {
        if(count == capacity) {
            std::size_t newCapacity = 2*capacity;
            void** newData = new void*[newCapacity];
            std::memcpy(newData, data, count*sizeof(void*));
            delete [] data;
            data = newData;
            capacity = newCapacity;
        }
        assert(count < capacity);
        data[count++] = t;
    }
    void pop() {
        assert(count > 0);
        --count;
    }
    void* top() const {
        assert(count > 0);
        return data[count-1];
    }
    std::size_t size() const { return count; }
};

// Partial specialization for other pointer types
template<class T> class Stack<T*> : private Stack<void*> {
    typedef Stack<void*> Base;
public:
    void push(T* const & t) { Base::push(t); }
    void pop() { Base::pop(); }
    T* top() const { return static_cast<T*>(Base::top()); }
    std::size_t size() { return Base::size(); }
};
#endif // NOBLOAT_H ///:~

```

这个简单的栈能根据需要进行容量的扩充。**void\***特化通过**template <>**前缀的功效（就是说，模板参数列表为空）做成了一个优秀的完全特化版本。如前所述，在一个类模板的特化中必然要实现所有的成员函数。这个特征同样存在于所有其他指针类型的类模板的特化中。由于仅仅想用**Stack <void\*>**作为实现目标，而且也不希望将它的任何接口直接暴露给用户，因此半特化只用于从**Stack <void\*>**私有派生出来的其他指针类型。每个指针实例化后的成员函数都是

**Stack<void\*>** 中相应函数的一个有细微改进的函数。因而，无论何时对一个非**void\***类型的指针类型进行实例化，它产生的代码只是单独使用基本（primary）模板所产生的代码的一小部分<sup>①</sup>。下面是一个驱动程序：

```
//: C05:NobloatTest.cpp
#include <iostream>
#include <string>
#include "Nobloat.h"
using namespace std;

template<class StackType>
void emptyTheStack(StackType& stk) {
    while(stk.size() > 0) {
        cout << stk.top() << endl;
        stk.pop();
    }
}

// An overload for emptyTheStack (not a specialization!)
template<class T>
void emptyTheStack(Stack<T*>& stk) {
    while(stk.size() > 0) {
        cout << *stk.top() << endl;
        stk.pop();
    }
}

int main() {
    Stack<int> s1;
    s1.push(1);
    s1.push(2);
    emptyTheStack(s1);
    Stack<int *> s2;
    int i = 3;
    int j = 4;
    s2.push(&i);
    s2.push(&j);
    emptyTheStack(s2);
} ///:~
```

为方便起见，在这个程序中包含了两个**emptyStack**函数模板。由于函数模板不支持半特化，所以程序中提供了重载的函数模板。**emptyStack**的第2个版本比第1个版本的特化程度更高一些，所以当用到指针类型特化函数模板的时候它总是被选用。在这个程序中实例化了3个类模板：**Stack<int>**、**Stack<void\*>** 和**Stack<int\*>**。由于**Stack<int\*>** 派生于**Stack<void\*>**，因此**Stack<void\*>** 是一种隐式实例化。如果一个程序要为多个指针类型进行实例化就可能产生大量的代码，可以通过一个**Stack**模板来节省大量的代码空间。

## 5.4 名称查找问题

当编译器碰到一个标识符时，它必须能够确定这个标识符所代表的实体的类型和作用域（如果它是一个变量，就是生存期）。模板的引入增加了这个问题的复杂度。当编译器首次看到一个模板定义时它不知道有关这个模板的任何信息，只有当它看到模板的实例化时，它才能判断这个模板是否被正确地使用了。这种状况导致了模板编译需要分两个阶段进行。

### 5.4.1 模板中的名称

在第1阶段，编译器解析模板定义，寻找明显的语法错误，还要对它所能解析的所有名称

<sup>①</sup> 由于这个改进（forwarding）函数是内联的，因此不产生**Stack<void\*>**的代码！

进行解析。对于不依赖于模板参数的名称，编译器使用普通名称查找的方法解析它们，如果有必要，编译器也会依赖模板参数进行查找（在后面讨论）。它不能够解析的名称就是所谓的关联名（dependent name），这些名称以某种方式依赖于模板参数。只有等到用实际参数来实例化模板的时候，这些名称才能被解析。因此模板编译的第2个阶段就是模板实例化。在这里，由编译器来决定是否使用模板的一个显式特化来取代基本的模板。

在看下面的例子之前，必须至少理解两个术语。限定名（qualified name）是指具有类名前缀，或者是被一个对象名加上点运算符修饰，或者是被一个指向某一对象的指针加一个箭头运算符所限定的名称修饰。限定名举例如下：

```
MyClass::f();
x.f();
p->f();
```

本教材中多次使用限定名，最近的用法是把它与**typename**关键字相联系。之所以称为限定名是因为这些目标名（如上面例子中的**f**）被明确的与一个类或者与一个名字空间相联系，它们会告诉编译器应该去哪儿寻找这些名称的声明。

另一个术语是关联参数查找（argument-dependent lookup<sup>⊖</sup>，ADL），这个机制起初是设计用来简化在名字空间中声明的非成员函数调用（包含运算符）。看下面的代码：

```
#include <iostream>
#include <string>
// ...
std::string s("hello");
std::cout << s << std::endl;
```

请注意，在头文件中的典型习惯用法中，没有使用**using namespace std**指令。没有了这条指令，就必须使用“**std::**”来限定**std**名字空间中的每项内容。但是，在这里并没有用它来限定**std**中的所有内容。你能知道哪一个是不合格的吗？

在程序中没有指定使用哪一个运算符函数。程序员希望下述事情发生，但却不想键入这些代码：

```
std::operator<<(std::operator<<(std::cout,s),std::endl);
```

为了使最初的输出语句能够按照预先的设计执行，ADL规定：当出现了对某个非限定函数的调用，而该非限定函数却没有在一个（标准）作用域内进行声明时，编译器为了匹配这个函数声明，就会寻找它的每一个参数的名字空间来进行匹配。在最初的语句中，第1个函数调用是：

```
operator<<(std::cout, s);
```

由于在初始引用的名字空间作用域中没有这个函数声明，编译器注意到这个函数的第1个参数（**std::cout**）在名字空间**std**中；因此它就把这个名字空间添加到作用域列表中，以此来寻找一个能完美匹配**operator<< (std::ostream&,std::string)**的独一无二的函数。它通过**<string>**头文件发现这个函数是在**std**名字空间中声明的。

没有ADL，名字空间的使用将会非常的不方便。注意，ADL通常从所有合格的名字空间中引入存有质疑的名称的所有声明——若没有一个最好的匹配，将会产生二义性。

为了避开ADL不用，可以将函数名称置于一对圆括号中：

```
(f)(x, y); // ADL suppressed
```

⊖ 也称为Koenig查找，因为Andrew Koenig首先向C++标准委员会建议了这种查找技术。ADL用一般概念阐述了模板是否应该包括于其中。

现在来看看下面的程序：<sup>①</sup>

```
//: C05:Lookup.cpp
// Only produces correct behavior with EDG,
// and Metrowerks using a special option.
#include <iostream>
using std::cout;
using std::endl;

void f(double) { cout << "f(double)" << endl; }

template<class T> class X {
public:
    void g() { f(1); }
};

void f(int) { cout << "f(int)" << endl; }

int main() {
    X<int>().g();
} ///:~
```

本程序使用的编译器是支持前端兼容用法的Edison Design Group (EDG)<sup>②</sup>，上面的代码在这个编译器上不用修改就能正确执行。某些编译器，例如Metrowerks，可以通过配置选项实现正确的查找。输出结果应该是：

```
f(double)
```

这是因为**f**是一个非关联的名称，它早在定义模板的过程中就已经解析了，那时只有**f(double)**在模板的作用域之中。遗憾的是，在实际的应用系统中存在着大量的依赖非标准行为的不规范代码，即将**g()**中**f(1)**的调用与其后的**f(int)**绑定在了一起，因此编译器的编写者也就不情愿的去做改动了。

下面是一个更详细的例子：<sup>③</sup>

```
//: C05:Lookup2.cpp {-bor}{-g++}{-dmc}
// Microsoft: use option -Za (ANSI mode)
#include <algorithm>
#include <iostream>
#include <typeinfo>
using std::cout;
using std::endl;

void g() { cout << "global g()" << endl; }

template<class T> class Y {
public:
    void g() {
        cout << "Y<" << typeid(T).name() << ">::g()" << endl;
    }
    void h() {
        cout << "Y<" << typeid(T).name() << ">::h()" << endl;
    }
    typedef int E;
};
```

① 这是Herb Sutter奉献的一个程序。

② 很多编译器使用这种前端兼容用法，包括Comeau C++。

③ 这也是基于Herb Sutter的一个例子。





```

};

typedef double E;

template<class T> void swap(T& t1, T& t2) {
    cout << "global swap" << endl;
    T temp = t1;
    t1 = t2;
    t2 = temp;
}

template<class T> class X : public Y<T> {
public:
    E f() {
        g();
        this->h();
        T t1 = T(), t2 = T(1);
        cout << t1 << endl;
        swap(t1, t2);
        std::swap(t1, t2);
        cout << typeid(E).name() << endl;
        return E(t2);
    }
};

int main() {
    X<int> x;
    cout << x.f() << endl;
} ///:~

```

这个程序的输出应该是：

```

global g()
Y<int>::h()
0
global swap
double
1

```

看看**X::f()**中的声明：

- **E**，是**X::f()**的返回类型，它不是一个关联名称，因此在解析模板的时候它就被找到了。编译器找到了**E**后，用**typedef**将**E**命名成一个**double**类型。这种情况可能看起来有点奇怪，因为在非模板类中，**E**在基类中的声明应该先被找到，但那是非模板类的规则。（基类**Y**，是一个关联基类（dependent base class），因此在模板定义期间不能够找到它）。
- **g()**的调用也是不依赖参数类型的，因为它没有用到**T**。若**g**带有某些定义在另一个名字空间中的类类型的参数，则ADL就会将这个名字空间接管过来，因为在它的作用域内没有带有这种参数的**g**定义。因此，这个调用匹配了**g()**的全局声明。
- **this->h()**调用是一个限定名称调用，限定它的对象（**this**）指的是当前对象，即该当前对象是**X**类型的，**X**通过继承机制又依赖于名称**Y<T>**。**X**中没有函数**h()**，因此查找将去**X**的基类**Y<T>**作用域内寻找。由于这是一个关联名称，它在实例化期间进行查找，**Y<T>**此时已经完全准确地知道（包括可能在**X**的定义之后编写的任何可能的特化）。因此它调用的是**Y<int>::h()**。
- **t1**和**t2**的声明是关联的。
- 对**operator<<(cout,t1)**的调用也是关联的，因为**t1**的类型为**T**。它是在**T**已经确定为**int**后才进行查找，并且在**std**中找到后添加了**int**。

- **swap( )**的非限定调用也是关联的，因为它的参数是类型**T**。这从根本上引起了全局**swap(int&, int&)**的实例化。
- **std::swap( )**的限定调用不是关联的，因为**std**是一个固定的名字空间。编译器知道去**std**中寻找合适的声明。“**::**”左边的限定词必须为关联的限定名称提及一个模板参数)。之后**std::swap( )**函数模板在实例化期间生成**std::swap(int&, int&)**。**X<T>::f( )**中再也没有关联名称了。

综上所述：若名称是关联的，则它的查找是在实例化时进行，非限定的关联名称除外，它是一个普通名称查找，它的查找进行的比较早是在定义时进行。所有模板中的非关联名称被较早地查找，这种查找是在模板定义被解析的时候进行。(若有必要，这种名称还有另一种实例化期间的查找，此时实际参数类型是已知的。)

如果读者已经理解了我们讨论过的例子，请准备好学习下一节有关**friend**声明的内容，它也许会带给读者另一个惊奇。

#### 5.4.2 模板和友元

在类中声明一个友元函数，就允许一个类的非成员函数访问这个类的非公有成员。若友元函数的名称是被限定的，则将会在限定它的名字空间或类中找到它。但是，如果它是非限定的，编译器必须假定在某处能找到这个友元函数的定义，因为所有的标识符必须有一个惟一的作用域。编程人员希望把这个函数定义在最近的封装名字空间（而非类）作用域内，这个作用域内也包括与那个函数有友元关系的类。通常这个作用域就是全局作用域。下面的非模板例子清晰地阐明了这一点：

```
//: C05:FriendScope.cpp
#include <iostream>
using namespace std;

class Friendly {
    int i;
public:
    Friendly(int theInt) { i = theInt; }
    friend void f(const Friendly&); // Needs global def.
    void g() { f(*this); }
};

void h() {
    f(Friendly(1)); // Uses ADL
}

void f(const Friendly& fo) { // Definition of friend
    cout << fo.i << endl;
}

int main() {
    h(); // Prints 1
    Friendly(2).g(); // Prints 2
} ///:~
```

**Friendly**类中**f( )**的声明是非限定的，因此编译器希望能最终将这个声明链接到位于文件作用域（在本例中就是包含**Friendly**的名字空间作用域）中的它的定义上。它的定义出现在函数**h( )**的定义之后。然而，**h( )**中对链接到同一函数的**f( )**的调用却是一件单独的事情。这个过程由ADL进行解析。由于**h( )**中的**f( )**的参数是一个**Friendly**对象，为了匹配**f( )**的声明，

编译器将会去寻找**Friendly**类，并将成功找到。如果调用的是**f(1)**（这是很有意义的，因为1能被隐含地转变为**Friendly(1)**），这个调用将会失败，因为没有任何提示来通知编译器应该去哪儿寻找这个**f()**的声明。在这种情况下，EDG编译器会正确地解释**f**没有定义。

现在假定**Friendly**和**f**都是模板，程序如下所示：

```
//: C05:FriendScope2.cpp
#include <iostream>
using namespace std;

// Necessary forward declarations:
template<class T> class Friendly;
template<class T> void f(const Friendly<T>&);

template<class T> class Friendly {
    T t;
public:
    Friendly(const T& theT) : t(theT) {}
    friend void f<>(const Friendly<T>&);
    void g() { f(*this); }
};

void h() {
    f(Friendly<int>(1));
}

template<class T> void f(const Friendly<T>& fo) {
    cout << fo.t << endl;
}

int main() {
    h();
    Friendly<int>(2).g();
} ///:~
```

首先要注意到**Friendly**中**f**的声明里的尖括号。这是必要的，它告诉编译器**f**是一个模板。否则，编译器就会去寻找一个名为**f**的普通函数而不会找到它。读者可能会在尖括号里加上模板参数（<T>），但它其实可以很容易地从声明中推断出来。

在类定义之前，提前声明函数模板**f**是很有必要的，虽然在前面的例子中并没有这个声明，因为当时**f**不是模板；这句话的意思清楚表明：友元函数模板必须提前声明。为了恰当地声明**f**，**Friendly**也必须在它之前进行声明，因为**f**具有一个**Friendly**参数，因此**Friendly**的声明在程序开始的最前面。也可以将**f**的完全定义放在**Friendly**的初始声明之后，这样就避免了将它的定义和声明分离开，但选择这样做是为了更接近上一个例子的格式。

为了在模板中使用友元，最后还可以选择这样做：在主类模板定义中完全地定义友元函数。下面来看看上例在这种情况下是如何修改的：

```
//: C05:FriendScope3.cpp {-bor}
// Microsoft: use the -Za (ANSI-compliant) option
#include <iostream>
using namespace std;

template<class T> class Friendly {
    T t;
public:
    Friendly(const T& theT) : t(theT) {}
    friend void f(const Friendly<T>& fo) {
        cout << fo.t << endl;
    }
}
```

```

    void g() { f(*this); }
};

void h() {
    f(Friendly<int>(1));
}

int main() {
    h();
    Friendly<int>(2).g();
} ///:~

```

本例和前面的例子有一个重要的区别：在这里 **f** 不再是一个模板，而是一个普通函数。（请记住，要表明 **f()** 是一个模板，尖括号是必不可少的。）**Friendly** 类模板每次实例化的时候，就会生成一个新的带有当前 **Friendly** 特化参数的普通重载函数。这就是为什么 Dan Saks 称之为“产生新友元”<sup>①</sup> 的原因。这是为模板定义友元函数的最方便的方式。

为了说明得更清楚一些，假设现在要想向一个类模板中加入非成员友元运算符。下面只是个仅持有一个普通值的类模板：

```

template<class T> class Box {
    T t;
public:
    Box(const T& theT) : t(theT) {}
};

```

有些初学者没有理解本节前面的例子，他们可能会灰心丧气。因为程序中没有一个简单的流输出插入符来验证程序所做的工作。如果不在 **Box** 的定义中定义自己的运算符，就必须提供早些时候讨论过的前置声明：

```

//: C05:Box1.cpp
// Defines template operators.
#include <iostream>
using namespace std;

// Forward declarations
template<class T> class Box;

template<class T>
Box<T> operator+(const Box<T>&, const Box<T>&);
template<class T>
ostream& operator<<(ostream&, const Box<T>&);

template<class T> class Box {
    T t;
public:
    Box(const T& theT) : t(theT) {}
    friend Box operator+<>(const Box<T>&, const Box<T>&);
    friend ostream& operator<< <>(ostream&, const Box<T>&);
};

template<class T>
Box<T> operator+(const Box<T>& b1, const Box<T>& b2) {
    return Box<T>(b1.t + b2.t);
}

template<class T>

```

① 来源于2001年9月份在波特兰的一次“C++研讨会”。

```
ostream& operator<<(ostream& os, const Box<T>& b) {
    return os << '[' << b.t << ']';
}

int main() {
    Box<int> b1(1), b2(2);
    cout << b1 + b2 << endl; // [3]
    // cout << b1 + 2 << endl; // No implicit conversions!
} ///:~
```

程序在这里定义了一个加运算符和一个输出流操作符。主程序揭示了这个方法的一个缺陷：无法使用隐式转换（如表达式**b1+2**），因为模板没有提供这些转换。使用内部类，非模板方法将会使程序变得短小、更强健：

```
//: C05:Box2.cpp
// Defines non-template operators.
#include <iostream>
using namespace std;

template<class T> class Box {
    T t;
public:
    Box(const T& theT) : t(theT) {}
    friend Box<T> operator+(const Box<T>& b1,
                           const Box<T>& b2) {
        return Box<T>(b1.t + b2.t);
    }
    friend ostream&
    operator<<(ostream& os, const Box<T>& b) {
        return os << '[' << b.t << ']';
    }
};

int main() {
    Box<int> b1(1), b2(2);
    cout << b1 + b2 << endl; // [3]
    cout << b1 + 2 << endl; // [3]
} ///:~
```

由于运算符成员函数是普通函数（为**Box**的每一个特化进行的重载——本例中就是**int**），像平常一样，隐式转换也可以使用；因此表达式**b1+2**是合法的。

注意，有一个特殊的类型不能成为**Box**或其他任意类模板的友元，这个类型是**T**——或由**T**参数化的类模板类型。无论如何，找不到一个很合理的原因解释为什么不能这样用，但的确如此，**friend class T**这个声明是不合法的，也不能被编译。

#### 友元模板

在程序中可以更精确地说明一个模板的哪些特化是类的友元。在上节的例子中，只有函数模板**f**是一个友元，它与特化**Friendly**的类型相同。举例来说，只有特化**f<int>(const Friendly<int>&)**才是类**Friendly<int>**的一个友元。这个例子是通过**Friendly**的模板参数来特化在其友元声明中的**f**的方式来实现的。若愿意，可以产生一个特别的、固定的**f**特化作为所有**Friendly**实例的一个友元，如下所示：

```
// Inside Friendly:
friend void f<>(const Friendly<double>&);
```

通过用**double**代替**T**，**f**的**double**特化可以访问任意**Friendly**特化的非公有成员。而**f<double>()**特化直到被明确调用时才会被实例化。

同样，若声明了一个参数不依赖于**T**的非模板函数，这个函数就是所有**Friendly**实例的一个友元：

```
// Inside Friendly:
friend void g(int); // g(int) befriends all Friendlys
```

同前面一样，由于**g(int)**是非限定的，他必须在文件作用域（包含**Friendly**的名字空间作用域）内定义。

也可以让**f**的所有特化成为所有**Friendly**特化的友元。只需要通过一个所谓的友元模板（**friend template**）来实现，如下所示：

```
template<class T> class Friendly {
    template<class U> friend void f<>(const Friendly<U>&);
```

由于友元声明的模板参数独立于**T**，因此任意的**T**和**U**的组合都允许使用，形成友元关系。像成员模板一样，友元模板也可以出现在非模板类中。

## 5.5 模板编程中的习语

语言是表达思想的一种工具，新的编程语言特征总是会产生新的程序设计技术。本节讨论一些经常使用的模板编程用语，自模板被引入C++语言，这些用语就已经出现并应用了好多年了。<sup>①</sup>

### 5.5.1 特征

特征模板技术，最先由Nathan Myers倡导，它是一种将与某种类型相关联的所有声明绑定在一起的实现方式。本质上说，使用特征技术，可以以一种灵活的方法从它们的语境中将这些类型和值进行“混合与匹配”，同时又使得程序的代码灵活易读并且易于维护。

一个最简单的特征模板的例子是定义在**<limits>**中的**numeric\_limits**类模板。这个基本模板的定义如下：

```
template<class T> class numeric_limits {
public:
    static const bool is_specialized = false;
    static T min() throw();
    static T max() throw();
    static const int digits = 0;
    static const int digits10 = 0;
    static const bool is_signed = false;
    static const bool is_integer = false;
    static const bool is_exact = false;
    static const int radix = 0;
    static T epsilon() throw();
    static T round_error() throw();
    static const int min_exponent = 0;
    static const int min_exponent10 = 0;
    static const int max_exponent = 0;
    static const int max_exponent10 = 0;
    static const bool has_infinity = false;
    static const bool has_quiet_NaN = false;
    static const bool has_signaling_NaN = false;
    static const float_denorm_style has_denorm =
        denorm_absent;
    static const bool has_denorm_loss = false;
```

① 另一个模板用语，混入继承，将在第9章讨论。



```

static T infinity() throw();
static T quiet_NaN() throw();
static T signaling_NaN() throw();
static T denorm_min() throw();
static const bool is_iec559 = false;
static const bool is_bounded = false;
static const bool is_modulo = false;
static const bool traps = false;
static const bool tinyness_before = false;
static const float_round_style round_style =
                                round_toward_zero;
};

```

**<limits>**头文件为所有基本数字类型定义了特化（当**is\_specialized**成员被设为**true**时）。例如，若想得到浮点数字系统的**double**版本的基类型，可以使用表达式**numeric\_limits<double>::radix**。为了得到有用的最小整数值，可以使用**numeric\_limits<int>::min()**。在程序中，并非向所有的**numeric\_limits**成员都提供了基本类型。（例如，**epsilon()**只对浮点数类型有意义。）

有些值总是整数，它们是**numeric\_limits**的静态数据成员。有些可能不总是整数值，例如**float**的最小值，它们作为静态内联成员函数实现。这是因为C++只允许在类定义中初始化整数（integral）静态数据成员常量。

在第3章中读者看到了字符串类如何使用特征技术控制字符处理函数。**std::string**类和**std::wstring**类是**std::basic\_string**模板的特化，它的定义如下所示：

```

template<class charT,
        class traits = char_traits<charT>,
        class allocator = allocator<charT> >
class basic_string;

```

模板参数**charT**代表了基础字符类型，它通常是**char**类型或**wchar\_t**类型。基本的**char\_traits**模板是典型的空模板，标准库提供了对**char**和**wchar\_t**进行的特化。下面是根据C++标准提供的一个**char\_traits<char>**特化：

```

template<> struct char_traits<char> {
    typedef char char_type;
    typedef int int_type;
    typedef streamoff off_type;
    typedef streampos pos_type;
    typedef mbstate_t state_type;
    static void assign(char_type& c1, const char_type& c2);
    static bool eq(const char_type& c1, const char_type& c2);
    static bool lt(const char_type& c1, const char_type& c2);
    static int compare(const char_type* s1,
                      const char_type* s2, size_t n);
    static size_t length(const char_type* s);
    static const char_type* find(const char_type* s,
                                size_t n,
                                const char_type& a);
    static char_type* move(char_type* s1,
                           const char_type* s2, size_t n);
    static char_type* copy(char_type* s1,
                           const char_type* s2, size_t n);
    static char_type* assign(char_type* s, size_t n,
                             char_type a);
    static int_type not_eof(const int_type& c);
    static char_type to_char_type(const int_type& c);
    static int_type to_int_type(const char_type& c);
    static bool eq_int_type(const int_type& c1,

```



```

        const int_type& c2);
    static int_type eof();
};

```

**basic\_string**类模板使用这些函数，用于基于字符操作的通用的字符串处理。当声明一个**string**变量时，例如：

```
std::string s;
```

事实上，正在声明的**s**格式如下所示（由于在**basic\_string**特化中有默认模板参数）：

```
std::basic_string<char, std::char_traits<char>,
    std::allocator<char> > s;
```

由于字符特征已经从**basic\_string**类模板中分离出来，可以使用一个惯用的特征类来取代**std::char\_traits**。下面的例子显示了这种灵活性：

```

//: C05: BearCorner.h
#ifndef BEARCORNER_H
#define BEARCORNER_H
#include <iostream>
using std::ostream;

// Item classes (traits of guests):
class Milk {
public:
    friend ostream& operator<<(ostream& os, const Milk&) {
        return os << "Milk";
    }
};

class CondensedMilk {
public:
    friend ostream&
    operator<<(ostream& os, const CondensedMilk &) {
        return os << "Condensed Milk";
    }
};

class Honey {
public:
    friend ostream& operator<<(ostream& os, const Honey&) {
        return os << "Honey";
    }
};

class Cookies {
public:
    friend ostream& operator<<(ostream& os, const Cookies&) {
        return os << "Cookies";
    }
};

// Guest classes:
class Bear {
public:
    friend ostream& operator<<(ostream& os, const Bear&) {
        return os << "Theodore";
    }
};

class Boy {
public:
    friend ostream& operator<<(ostream& os, const Boy&) {

```



```

        return os << "Patrick";
    }
};

// Primary traits template (empty-could hold common types)
template<class Guest> class GuestTraits;

// Traits specializations for Guest types
template<> class GuestTraits<Bear> {
public:
    typedef CondensedMilk beverage_type;
    typedef Honey snack_type;
};

template<> class GuestTraits<Boy> {
public:
    typedef Milk beverage_type;
    typedef Cookies snack_type;
};

#endif // BEARCORNER_H ///:~

//: C05: BearCorner.cpp
// Illustrates traits classes.
#include <iostream>
#include "BearCorner.h"
using namespace std;

// A custom traits class
class MixedUpTraits {
public:
    typedef Milk beverage_type;
    typedef Honey snack_type;
};

// The Guest template (uses a traits class)
template<class Guest, class traits = GuestTraits<Guest> >
class BearCorner {
    Guest theGuest;
    typedef typename traits::beverage_type beverage_type;
    typedef typename traits::snack_type snack_type;
    beverage_type bev;
    snack_type snack;
public:
    BearCorner(const Guest& g) : theGuest(g) {}
    void entertain() {
        cout << "Entertaining " << theGuest
              << " serving " << bev
              << " and " << snack << endl;
    }
};

int main() {
    Boy cr;
    BearCorner<Boy> pc1(cr);
    pc1.entertain();
    Bear pb;
    BearCorner<Bear> pc2(pb);
    pc2.entertain();
    BearCorner<Bear, MixedUpTraits> pc3(pb);
    pc3.entertain();
} ///:~

```

在这个程序中，为招待作为客人的类**Boy**和类**Bear**的实例，提供了适合他们口味的食物。

**Boy**喜欢牛奶和小甜点，**Bear**喜欢浓缩的牛奶和蜂蜜。客人与食物之间的关联是通过一个基本的（空的）特征类模板的特化完成的。**BearCorner**的默认参数保证了客人能够获得恰当的食物，但也可以用一个简单的符合特征类需求的类来代替它，就像上面用到的**MixedUpTraits**类。这个程序的输出是：

```
Entertaining Patrick serving Milk and Cookies
Entertaining Theodore serving Condensed Milk and Honey
Entertaining Theodore serving Milk and Honey
```

特征类的使用提供了两个关键的优点：（1）在将对象与其关联的属性或函数配对方面提供了灵活性和可扩充性；（2）它保持了模板参数列表的短小易读。如果一个客人与30个类型相关，那么，将所有30个参数直接在每一个**BearCorner**声明中指定，这将是极不方便的。而将这些类型放在一个独立的特征类中就会大大简化这项工作。

如第4章所述，特征技术也可用于实现流和区域化。在第6章有一个名为**PrintSequence.h**头文件，其中可以找到一个迭代器特征类的例子。

### 5.5.2 策略

如果检查一下用**wchar\_t**特化的**char\_traits**，就会发现实际上它相当于**char**特化的副本：

```
template<> struct char_traits<wchar_t> {
    typedef wchar_t char_type;
    typedef wint_t int_type;
    typedef streamoff off_type;
    typedef wstreampos pos_type;
    typedef mbstate_t state_type;
    static void assign(char_type& c1, const char_type& c2);
    static bool eq(const char_type& c1, const char_type& c2);
    static bool lt(const char_type& c1, const char_type& c2);
    static int compare(const char_type* s1,
                      const char_type* s2, size_t n);
    static size_t length(const char_type* s);
    static const char_type* find(const char_type* s,
                                size_t n,
                                const char_type& a);
    static char_type* move(char_type* s1,
                           const char_type* s2, size_t n);
    static char_type* copy(char_type* s1,
                           const char_type* s2, size_t n);
    static char_type* assign(char_type* s, size_t n,
                             char_type a);
    static int_type not_eof(const int_type& c);
    static char_type to_char_type(const int_type& c);
    static int_type to_int_type(const char_type& c);
    static bool eq_int_type(const int_type& c1,
                             const int_type& c2);
    static int_type eof();
};
```

两个版本惟一的真正的区别是，所包含的类型集不同（**char**和**int**分别相对于**wchar\_t**和**wint\_t**）。两者所提供的函数是相同的。<sup>①</sup>这更突出了一个事实：特征类是为特征（trait）而设计的，在相关的特征类之间的改变通常就是类型和常量值，或者是使用了相关类型的模板参数的固定算法。通常特征类本身就是模板，因为它们包含的类型和常量通常被看做是基本模

<sup>①</sup> 实际上（这不是实质上的。例如，）**char\_traits<>::compare()** 在一个实例中可能调用函数**strcmp()**，而在另一个实例中就有可能调用**wcscmp()**。而在这里所说的是**compare()**函数执行的功能是相同的。

板的特征参数（例如，**char**和**wchar\_t**）。

将函数（functionality）与模板参数关联起来也是有用的，因而客户端程序员在他们编码的时候能够轻松地定制代码行为。举例来说，下面的这个**BearCorner**程序版本，支持不同的招待类型：

```
//: C05: BearCorner2.cpp
// Illustrates policy classes.
#include <iostream>
#include "BearCorner.h"
using namespace std;

// Policy classes (require a static doAction() function):
class Feed {
public:
    static const char* doAction() { return "Feeding"; }
};
class Stuff {
public:
    static const char* doAction() { return "Stuffing"; }
};

// The Guest template (uses a policy and a traits class)
template<class Guest, class Action,
        class traits = GuestTraits<Guest> >
class BearCorner {
    Guest theGuest;
    typedef typename traits::beverage_type beverage_type;
    typedef typename traits::snack_type snack_type;
    beverage_type bev;
    snack_type snack;
public:
    BearCorner(const Guest& g) : theGuest(g) {}
    void entertain() {
        cout << Action::doAction() << " " << theGuest
              << " with " << bev
              << " and " << snack << endl;
    }
};

int main() {
    Boy cr;
    BearCorner<Boy, Feed> pc1(cr);
    pc1.entertain();
    Bear pb;
    BearCorner<Bear, Stuff> pc2(pb);
    pc2.entertain();
} ///:-
```

**BearCorner**类中的**Action**模板参数希望有一个名为**doAction()**的静态成员函数，它用在**BearCorner<>::entertain()**中。用户按照意愿可以选择**Feed**或**Stuff**，二者都提供了所需的函数。用这种方式来封装函数的类称为策略类（policy class）。在上例中，招待“策略”是通过**Feed::doAction()**和**Stuff::doAction()**提供的。这些策略类可能是普通类，也可能是模板，还有可能是结合了使用继承机制全部优点的类。关于基于策略的更深入的设计技术，请参看Andrei Alexandrescu的书<sup>①</sup>。关于这个主题，这本书具有权威性。

① 《Modern C++ Design: Generic Programming and Design Patterns Applied》，Addison Wesley, 2001。

### 5.5.3 奇特的递归模板模式

任何一个初学C++的程序设计者都知道如何修改一个类，使它跟踪一个类当前实际存在的对象个数。必须做的所有工作就是添加静态成员、修改构造函数和析构函数的逻辑，如下所示：

```

//: C05:CountedClass.cpp
// Object counting via static members.
#include <iostream>
using namespace std;

class CountedClass {
    static int count;
public:
    CountedClass() { ++count; }
    CountedClass(const CountedClass&) { ++count; }
    ~CountedClass() { --count; }
    static int getCount() { return count; }
};

int CountedClass::count = 0;

int main() {
    CountedClass a;
    cout << CountedClass::getCount() << endl;    // 1
    CountedClass b;
    cout << CountedClass::getCount() << endl;    // 2
    { // An arbitrary scope:
        CountedClass c(b);
        cout << CountedClass::getCount() << endl; // 3
        a = c;
        cout << CountedClass::getCount() << endl; // 3
    }
    cout << CountedClass::getCount() << endl;    // 2
} ///:~

```

**CountedClass**的所有构造函数都对静态数据成员**count**进行增1计数，而析构函数进行减1计数。静态成员函数**getCount()**获取当前对象的个数。

每次想为新添加的一个类的对象进行计数的时候，手工添加这些成员实在是太枯燥了。在面向对象程序设计中，过去常常对代码进行重用或共享采用的是继承方式，在本例中这也只是半个解决方案。请观察，当在基类中使用计数逻辑时会有什么情况发生：

```

//: C05:CountedClass2.cpp
// Erroneous attempt to count objects.
#include <iostream>
using namespace std;

class Counted {
    static int count;
public:
    Counted() { ++count; }
    Counted(const Counted&) { ++count; }
    ~Counted() { --count; }
    static int getCount() { return count; }
};

int Counted::count = 0;

class CountedClass : public Counted {};
class CountedClass2 : public Counted {};

int main() {

```



```

    CountedClass a;
    cout << CountedClass::getCount() << endl;    // 1
    CountedClass b;
    cout << CountedClass::getCount() << endl;    // 2
    CountedClass2 c;
    cout << CountedClass2::getCount() << endl;    // 3 (Error)
} ///:~

```

派生自**Counted**的所有类都共享了相同的、惟一的静态数据成员，因此通过跨越**Counted**层次结构中所有的类，它们的对象个数全部被跟踪。现在所需的是，有一种能自动为每个派生类生成一个不同基类的方式。一种奇特的模板构造实现了这种方式，如下所示：

```

//: C05:CountedClass3.cpp
#include <iostream>
using namespace std;

template<class T> class Counted {
    static int count;
public:
    Counted() { ++count; }
    Counted(const Counted<T>&) { ++count; }
    ~Counted() { --count; }
    static int getCount() { return count; }
};

template<class T> int Counted<T>::count = 0;

// Curious class definitions
class CountedClass : public Counted<CountedClass> {};
class CountedClass2 : public Counted<CountedClass2> {};

int main() {
    CountedClass a;
    cout << CountedClass::getCount() << endl;    // 1
    CountedClass b;
    cout << CountedClass::getCount() << endl;    // 2
    CountedClass2 c;
    cout << CountedClass2::getCount() << endl;    // 1 (!)
} ///:~

```

每个派生类都派生于一个惟一的基类，这个基类将它本身（派生类）作为模板参数！它看起来像是陷入了一个递归（循环）的定义，而且还有可能在某次计算中将某个任意的基类成员作为模板参数。由于**Counted**的数据成员不依赖于**T**，当模板被解析的时候，**Counted**的大小（为零！）就可以知道。因此究竟使用什么样的参数来实例化**Counted**无关紧要，因为它的大小总是相同的。当它被解析时，用任意一个**Counted**实例的派生类当然也可以完成，而且不会产生递归。由于每个基类都是惟一的，它有属于自己的静态数据，因而无论如何，这都是一个实现了向任意类中添加计数的便捷方法。Jim Coplien是第1个在刊物上提出这种有趣的派生方法的人；他在一篇名为“奇特的递归模板模式（curiously recurring template pattern）”<sup>①</sup>的文章中提出了这个方法。

## 5.6 模板元编程

1993年，编译器开始支持简单的模板构造，因此用户可以定义通用的容器和函数。同一时期，C++标准委员会也正在考虑将STL纳入标准C++，当时在C++标准委员会的成员们周围，

① 《C++Gems》，由Stan Lippman编辑，SIGS，1996。

到处都是那些已经通过验证的精巧的和令人惊讶的程序例子<sup>①</sup>，其中一个简单的例子如下所示：

```
//: C05:Factorial.cpp
// Compile-time computation using templates.
#include <iostream>
using namespace std;

template<int n> struct Factorial {
    enum { val = Factorial<n-1>::val * n };
};

template<> struct Factorial<0> {
    enum { val = 1 };
};

int main() {
    cout << Factorial<12>::val << endl; // 479001600
} ///:~
```

程序输出了由参数**12!**实例化后的正确值！并没有警告发出。那么警告是什么呢？警告就是：在程序开始运行前就已经完成了计算！

当编译器试图对**Factorial<12>**进行实例化时，编译器发现必须先实例化**Factorial<11>**，而后者又要求实例化**Factorial<10>**，以此类推。这个递归最终在特化**Factorial<1>**时结束，此时计算展开，**Factorial<12>::val**由整数常量479 001 600代替，至此编译结束。由于所有的计算都由编译器来做，其包含的值必须是编译时常量，因此使用了**enum**。程序运行时，惟一要做的工作就是跟随一个换行符来打印这个常量的值。为了说服自己，使读者相信是**Factorial**的一个特化导致了产生正确的编译时结果值，可以用它作为一个数组的维数来验证一下，如下所示：

```
double nums[Factorial<5>::val];
assert(sizeof nums == sizeof(double)*120);
```

### 5.6.1 编译时编程

正如将进行类型参数代替作为一种方便的方法，这意味着产生了一种支持编译时编程的机制。这样的程序称为**模板元程序**（template metaprogram）（因为正在“为一个程序进行编程”），事实证明可以用它做很多的事情。实际上，模板元编程就是完全的图灵机（Turing complete），因为它支持选择（if-else）和循环（通过递归）。从理论上讲，可以用它执行任何的计算<sup>②</sup>。上面的**factorial**程序例子显示了如何实现循环：编写一个递归模板，并且通过一个特化来提供一个终止递归的规则。下面的例子显示了如何利用相同的技术在编译时计算斐波那契（Fibonacci）数：

```
//: C05:Fibonacci.cpp
#include <iostream>
using namespace std;

template<int n> struct Fib {
    enum { val = Fib<n-1>::val + Fib<n-2>::val };
};
```

① 技术上讲，这些都是编译时常值，因此可能会说其中的标识符按照习惯用法应该全是大写字母，之所以坚持用小写是因为在这里它们模拟了变量。

② 1966年，Böhm和Jacopini证明了任意具有以下特征的语言都相当于一个图灵机：支持选择和循环，并具有使用变量随机数的能力。图灵机被认为具有表达任意算法的能力。

```

template<> struct Fib<1> { enum { val = 1 }; };

template<> struct Fib<0> { enum { val = 0 }; };

int main() {
    cout << Fib<5>::val << endl;    // 6
    cout << Fib<20>::val << endl;    // 6765
} ///:~

```

斐波那契数的数学定义如下：

$$f_n = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ f_{n-2} + f_{n-1}, & n > 1 \end{cases}$$

前两种情况导致了上面的模板特化，第3行的规则就是基本的模板。

#### 1. 编译时循环

在一个模板元程序中要计算任意的循环，首先必须再用公式表示递归。例如，若想计算整数  $n$  的  $p$  次方，下面几行程序使用了一个循环就可以完成：

```

int val = 1;
while(p-->0)
    val *= n;

```

也可以将它写成一个递归过程：

```

int power(int n, int p) {
    return (p == 0) ? 1 : n*power(n, p - 1);
}

```

现在它就可以很容易地用一个模板元程序来实现：

```

//: C05:Power.cpp
#include <iostream>
using namespace std;

template<int N, int P> struct Power {
    enum { val = N * Power<N, P-1>::val };
};

template<int N> struct Power<N, 0> {
    enum { val = 1 };
};

int main() {
    cout << Power<2, 5>::val << endl;    // 32
} ///:~

```

由于  $N$  仍是一个自由模板参数，因此需要用一半特化来作为终止条件。注意，这个程序仅当指数为非负时才运行。

由Czarnecki和Eisenecker<sup>⊖</sup>改编的下述元程序是很有趣的，因为它使用一个模板作为模板参数，模拟传递一个函数作为另一个函数的参数。其中的“循环是通过”  $0..n$  这些数字来实现的：

⊖ Czarnecki 和 Eisenecker, 《Generative Programming: Methods, Tools, and Applications》, Addison Wesley, 2000, 第417页。

```

//: C05:Accumulate.cpp
// Passes a "function" as a parameter at compile time.
#include <iostream>
using namespace std;

// Accumulates the results of F(0)..F(n)
template<int n, template<int> class F> struct Accumulate {
    enum { val = Accumulate<n-1, F>::val + F<n>::val };
};

// The stopping criterion (returns the value F(0))
template<template<int> class F> struct Accumulate<0, F> {
    enum { val = F<0>::val };
};

// Various "functions":
template<int n> struct Identity {
    enum { val = n };
};

template<int n> struct Square {
    enum { val = n*n };
};

template<int n> struct Cube {
    enum { val = n*n*n };
};

int main() {
    cout << Accumulate<4, Identity>::val << endl; // 10
    cout << Accumulate<4, Square>::val << endl;    // 30
    cout << Accumulate<4, Cube>::val << endl;      // 100
} ///:~

```

基本的**Accumulate**模板试图计算 $F(n)+F(n-1)\dots F(0)$ 的和。终止递归是通过一个“返回” $F(0)$ 的半特化来实现的。参数**F**本身就是一个模板，在本节以前的例子中它通常是一个函数。模板**Identity**、**Square**和**Cube**，用它们的模板参数计算自己名字表明的相应函数。在**main()**函数中，**Accumulate**的第1个实例化计算是求和： $4+3+2+1+0$ ，因此**Identity**函数仅仅“返回”它的模板参数。**main()**中第2行是计算这些数字的平方和： $(16+9+4+1+0)$ 。最后计算立方和： $(64+27+8+1+0)$ 。

## 2. 循环分解

算法设计者们总是尽力优化他们的程序。其中一个是在时间方面的优化——特别是在数字计算编程中——采用的是循环分解 (loop unrolling)，这是一项将顶层循环的次数减到最小的技术。典型的循环分解的例子是矩阵相乘。下面的函数将一个矩阵与一个向量相乘（假设常量**ROWS**和**COLS**已经定义过了）：

```

void mult(int a[ROWS][COLS], int x[COLS], int y[COLS]) {
    for(int i = 0; i < ROWS; ++i) {
        y[i] = 0;
        for(int j = 0; j < COLS; ++j)
            y[i] += a[i][j]*x[j];
    }
}

```

如果**COLS**是偶数，则进行增1和比较循环控制变量**j**的那一层循环体，就能用将该计算“分解”的方法切成两部分，使其变成在内部循环中成对出现的两部分计算：

```

void mult(int a[ROWS][COLS], int x[COLS], int y[COLS]) {
    for(int i = 0; i < ROWS; ++i) {

```



```

        y[i] = 0;
        for(int j = 0; j < COLS; j += 2)
            y[i] += a[i][j]*x[j] + a[i][j+1]*x[j+1];
    }
}

```

通常,若**COLS**是**k**的一个因子,每次内部循环迭代都可以执行**k**操作,大量减少顶层的循环。这种节省只有在大数组上操作才是明显的,但这正好是一个产业的强度数学计算的严谨的例子。

内联函数也构成了循环分解的一种格式。请看下面计算整数幂的方法:

```

//: C05:Unroll.cpp
// Unrolls an implicit loop via inlining.
#include <iostream>
using namespace std;

template<int n> inline int power(int m) {
    return power<n-1>(m) * m;
}

template<> inline int power<1>(int m) {
    return m;
}

template<> inline int power<0>(int m) {
    return 1;
}

int main() {
    int m = 4;
    cout << power<3>(m) << endl;
} ///:~

```

从概念上来讲,编译器必须生成模板参数分别为3、2、1的3个**power< >**特化。因为每个函数的代码可以内联,实际插入**main()**函数的代码就是一个单一的表达式**m\*m\*m**。这样一来,一个存在内联的简单模板特化就提供了一种方法,该方法可以完全避免循环控制顶层的出现<sup>①</sup>。这种循环分解的方法受使用的编译器的内联深度的限制。

### 3. 编译时选择

为了模拟在编译时的条件,可以在一个**enum**声明中使用3目条件运算符。下面的程序就使用了这个技术,在编译时计算两个整数之中的最大值:

```

//: C05:Max.cpp
#include <iostream>
using namespace std;

template<int n1, int n2> struct Max {
    enum { val = n1 > n2 ? n1 : n2 };
};

int main() {
    cout << Max<10, 20>::val << endl; // 20
} ///:~

```

如果想使用编译时条件来控制自定义代码的生成,可以利用**true**和**false**值的特化:

① 有一种更好的计算整数幂的方法: Russian Peasant算法。

```

//: C05:Conditionals.cpp
// Uses compile-time conditions to choose code.
#include <iostream>
using namespace std;

template<bool cond> struct Select {};

template<> class Select<true> {
    static void statement1() {
        cout << "This is statement1 executing\n";
    }
public:
    static void f() { statement1(); }
};

template<> class Select<false> {
    static void statement2() {
        cout << "This is statement2 executing\n";
    }
public:
    static void f() { statement2(); }
};

template<bool cond> void execute() {
    Select<cond>::f();
}

int main() {
    execute<sizeof(int) == 4>();
} ///:~

```

这个程序相当于下面的表达式:

```

if(cond)
    statement1();
else
    statement2();

```

除了条件`cond`在编译时确定之外, `execute<>()`和`Select<>`的适当版本都由编译器进行实例化。函数`Select<>::f()`在运行时执行。可以用类似的方式仿效一个`switch`语句, 但不是用值`true`和`false`, 而是特化每个case值。

#### 4. 编译时断言

在第2章中讨论了将断言(assertion)作为整个防御性编程策略的一部分的优点。断言基本上就是一个其后有一个适当动作的布尔表达式的判断; 若条件为真则什么都不做, 否则就停止并附带一个诊断消息。最好能尽快发现断言失败。若可以在编译时对一个表达式求值, 就使用编译时断言。下面的例子使用了这个技术, 它将一个布尔表达式映射到一个数组声明中:

```

//: C05:StaticAssert1.cpp {-xo}
// A simple, compile-time assertion facility

#define STATIC_ASSERT(x) \
    do { typedef int a[(x) ? 1 : -1]; } while(0)

int main() {
    STATIC_ASSERT(sizeof(int) <= sizeof(long)); // Passes
    STATIC_ASSERT(sizeof(double) <= sizeof(int)); // Fails
} ///:~

```

`do`循环为一个数组`a`的定义产生了一个临时空间, 这个数组的大小由一个不确定的条件所决定。定义一个大小为-1的数组是不合法的, 因此当条件为假时这条语句将失败。

前面的内容说明了如何对编译时布尔表达式求值。在效仿编译时断言方面剩下的问题就是打印一个有意义的错误消息并且停止编译。所有的编译错误都要求编译器停止编译；解决这个问题一个技巧是在错误消息中插入有用的文本。从Alexandrescu<sup>①</sup>处获得的下面的例子使用了模板特化，一个局部类和一个小巧且奇妙的宏来完成这项工作：

```
//: C05:StaticAssert2.cpp {-g++}
#include <iostream>
using namespace std;

// A template and a specialization
template<bool> struct StaticCheck {
    StaticCheck(...);
};

template<> struct StaticCheck<false> {};

// The macro (generates a local class)
#define STATIC_CHECK(expr, msg) { \
    class Error_##msg {}; \
    sizeof((StaticCheck<expr>(Error_##msg()))); \
}

// Detects narrowing conversions
template<class To, class From> To safe_cast(From from) {
    STATIC_CHECK(sizeof(From) <= sizeof(To),
        NarrowingConversion);
    return reinterpret_cast<To>(from);
}

int main() {
    void* p = 0;
    int i = safe_cast<int>(p);
    cout << "int cast okay" << endl;
    /// char c = safe_cast<char>(p);
} ///:~
```

这个例子定义了一个函数模板**safe\_cast<>()**用来进行两个对象长度的检查。它检查源对象类型长度是否不大于目标对象类型的长度。如果目标对象类型的长度较小，则用户将会在编译时得到通知：现正试图进行一个窄类型转换。注意，**StaticCheck**类模板有一个奇特的特性：任何模板参数的特化都可以被转换成**StaticCheck<true>**的实例（由于它的构造函数中的省略号<sup>②</sup>），并且没有任何模板参数的特化可以被转换成**StaticCheck<false>**的实例，因为没有为这种特化提供转换。它的思想是：在编译时如果相关条件在测试时为真，就创建一个新类的实例并且将它转换成为**StaticCheck<true>**对象；或者当条件在测试时为假，将它转换成一个**StaticCheck<false>**对象。由于**sizeof**运算符在编译时完成它的工作，因而用它来执行转换任务。当条件为假时，编译器将做出解释：它不知道如何将这个新类类型转换成**StaticCheck<false>**对象。（在**STATIC\_CHECK()**中的**sizeof**调用里面的特殊圆括号，是为了防止编译器认为程序正在试图将**sizeof**作为函数调用，这是不合法的）。为了在错误消息中插入一些有意义的信息，新的类名在它的名字中携带了关键且有意义的文字信息。

理解这项技术的最好的方式就是使其融入程序，请看上面例子**main()**中的这一行：

① 《Modern C++ Design》，第23~26页。

② 不允许将对象类型（除了内建的）传递给一个省略号参数特化，但是由于只是计算它的大小（一个编译时操作），实际上表达式在运行时没有被判断。

```
int i = safe_cast<int>(p);
```

**safe\_cast<int>(p)**的调用包含了下面的宏扩充代码，它代替了第1行代码：

```
{
    class Error_NarrowingConversion {};
    sizeof(StaticCheck<sizeof(void*) <= sizeof(int)> \
        (Error_NarrowingConversion()));
}
```

(回忆一下标记传递预处理操作符：**##**，它将它的操作数连接为一个单一标记，因此经过预处理后**Error\_##NarrowingConversion**变成了标记**Error\_NarrowingConversion**。) **Error\_NarrowingConversion**类是一个局部类（意味着它在一个非名字空间作用域内声明），因为它无需在这个程序的其他地方使用。这里**sizeof**运算符的使用试图决定**StaticCheck<true>**的实例的大小（因为在本程序使用的系统平台上**sizeof(void\*) <= sizeof(int)**为真），由**Error\_NarrowingConversion()**调用返回的临时对象隐式产生。编译器知道新类**Error\_NarrowingConversion**的大小（它为0），因此在编译时**sizeof**在**STATIC\_CHECK()**中外层的使用是合法的。由于从**Error\_NarrowingConversion**临时对象转换到**StaticCheck<true>**实例取得成功，因而这个**sizeof**的外层应用和执行都可以继续。

现在来看看，如果**main()**函数中最后一行的注解被去掉将会发生什么事情：

```
char c = safe_cast<char>(p);
```

在这里，把**safe\_cast<char>(p)**中的**STATIC\_CHECK()**宏扩充为：

```
{
    class Error_NarrowingConversion {};
    sizeof(StaticCheck<sizeof(void*) <= sizeof(char)> \
        (Error_NarrowingConversion()));
}
```

由于表达式**sizeof(void\*) <= sizeof(char)**为假，此时程序将尝试进行从**Error\_NarrowingConversion**临时对象到**StaticCheck<false>**实例的转换，如下所示：

```
sizeof(StaticCheck<false>(Error_NarrowingConversion()));
```

它失败了，所以编译器将发出一个如下的消息并停止工作：

```
Cannot cast from 'Error_NarrowingConversion' to
'StaticCheck<0>' in function
char safe_cast<char,void *>(void *)
```

类名**Error\_NarrowingConversion**是由编码人员巧妙设计的有意义的消息。通常为了用这种技术来执行一个静态断言，应该调用**STATIC\_CHECK**宏去进行编译时条件检查，并且用一个有意义的名称（函数名称、参数名称、模板名称等）来描述这个错误。

### 5.6.2 表达式模板

模板最强大的应用大概是在1994年由Todd Veldhuizen<sup>①</sup>和Daveed Vandevoorde<sup>②</sup>分别提出的一类模板技术：表达式模板（expression template）。表达式模板能够使某些计算得到的全方位

① 在Lippman的《C++ Gems》，SIGS，1996中可以找到Todd的原文的再版。它也表明了除了保留数学符号和优化的代码，表达式模板也允许在C++库中结合使用其他编程语言中的范例和机制，如Lambda表达式。另一个例子是奇特的类库Spirit，它是一个大量使用表达式模板的语法剖析器，它允许在C++中直接使用（一个近似的）EBNF符号，且产生了非常有效的语法剖析器。参看<http://spirit.sourceforge.net/>。

② 参看他他和Nico的《C++ Templates》，这是一本早期的权威著作。

的编译时优化, 这些优化产生这样一些代码, 其执行起来至少像支持优化的Fortran (专门用于科学计算的编程语言) 代码一样快速, 并且通过运算符重载仍旧保持了数学的原始符号。尽管可能在日常编程中并不使用这种技术, 但它是由C++编写的许多复杂的高性能数学库的基础<sup>①</sup>。

为了引发读者学习表达式模板的兴趣, 请看一个典型的数值线性代数的运算, 将两个矩阵或向量相加<sup>②</sup>, 如下所示:

```
D = A + B + C;
```

按照一般初学者的实现方式, 这个表达式将会导致一些临时变量的产生——一个是 $A+B$ , 一个是 $(A+B)+C$ 。当这些变量代表极大的矩阵或向量时, 这种方法将会耗尽系统的资源的确让人无法接受。表达式模板允许在没有临时变量的情况下使用同一个表达式。

下面的程序定义了一个**MyVector**类来模拟任意大小的数学向量。在程序中使用一个无类型的模板参数来表示向量的长度。程序还定义了一个**MyVectorSum**类来担当一个中间代理类, 用其计算**MyVector**对象之和。这将允许使用惰性计算, 因而向量的各个组成部分的相加不需要临时变量就可以执行。

```
//: C05:MyVector.cpp
// Optimizes away temporaries via templates.
#include <cstddef>
#include <cstdlib>
#include <ctime>
#include <iostream>
using namespace std;

// A proxy class for sums of vectors
template<class, size_t> class MyVectorSum;

template<class T, size_t N> class MyVector {
    T data[N];
public:
    MyVector<T,N>& operator=(const MyVector<T,N>& right) {
        for(size_t i = 0; i < N; ++i)
            data[i] = right.data[i];
        return *this;
    }
    MyVector<T,N>& operator=(const MyVectorSum<T,N>& right);
    const T& operator[](size_t i) const { return data[i]; }
    T& operator[](size_t i) { return data[i]; }
};

// Proxy class hold references; uses lazy addition
template<class T, size_t N> class MyVectorSum {
    const MyVector<T,N>& left;
    const MyVector<T,N>& right;
public:
    MyVectorSum(const MyVector<T,N>& lhs,
                const MyVector<T,N>& rhs)
        : left(lhs), right(rhs) {}
    T operator[](size_t i) const {
        return left[i] + right[i];
    }
};
```

① 即《Blitz++》(<http://www.oonumerics.org/blitz/>), 《the Matrix Template Library》(<http://www.osl.iu.edu/research/mtl/>) 和《POOMA》(<http://www.acl.lanl.gov/pooma/>)。

② 指的是在数学中的“向量”, 它是一个固定长度、一维的数值数组。

```

// Operator to support v3 = v1 + v2
template<class T, size_t N> MyVector<T,N>&
MyVector<T,N>::operator=(const MyVectorSum<T,N>& right) {
    for(size_t i = 0; i < N; ++i)
        data[i] = right[i];
    return *this;
}

// operator+ just stores references
template<class T, size_t N> inline MyVectorSum<T,N>
operator+(const MyVector<T,N>& left,
          const MyVector<T,N>& right) {
    return MyVectorSum<T,N>(left, right);
}

// Convenience functions for the test program below
template<class T, size_t N> void init(MyVector<T,N>& v) {
    for(size_t i = 0; i < N; ++i)
        v[i] = rand() % 100;
}

template<class T, size_t N> void print(MyVector<T,N>& v) {
    for(size_t i = 0; i < N; ++i)
        cout << v[i] << ' ';
    cout << endl;
}

int main() {
    srand(time(0));
    MyVector<int, 5> v1;
    init(v1);
    print(v1);
    MyVector<int, 5> v2;
    init(v2);
    print(v2);
    MyVector<int, 5> v3;
    v3 = v1 + v2;
    print(v3);
    MyVector<int, 5> v4;
    // Not yet supported:
    //! v4 = v1 + v2 + v3;
} ///:~

```

当**MyVectorSum**类产生时，它并不进行计算，它只是持有两个待加向量的引用。仅当访问一个向量中的成员（即它的**operator[]()**）时计算才会发生。为了对**MyVector**的赋值操作符进行重载，将**MyVectorSum**作为一个表达式的参数来使用，如下所示：

```
v1 = v2 + v3; // Add two vectors
```

当对表达式**v1+v2**求值时，返回一个**MyVectorSum**对象（实际上，是一个插入的内联对象，因为**operator+()**已经声明为**inline**）。这是一个很小的、固定大小的对象（它仅仅持有两个引用）。然后调用上面提到的赋值操作符：

```
v3.operator=(int,5)(MyVectorSum<int,5>(v2, v3));
```

这个运算采用实时运算的方式，将**v1**和**v2**的相应元素相加得到的和赋值给**v3**各自相应的元素。这不会产生**MyVector**的临时对象。

然而这个程序不支持多于两个操作数的表达式运算，比如

```
v4 = v1 + v2 + v3;
```

原因是在第1次相加后，还会尝试进行第2次相加：

```
(v1 + v2) + v3;
```

这个表达式需要一个重载运算符**operator+( )**，它的第1个参数是**MyVectorSum**类型，第2个参数是**MyVector**类型。可以尝试提供多个重载来满足所有的情况，但最好的办法是让模板来做这项工作，如下面的程序所示：

```
//: C05:MyVector2.cpp
// Handles sums of any length with expression templates.
#include <cstdint>
#include <cstdlib>
#include <ctime>
#include <iostream>
using namespace std;

// A proxy class for sums of vectors
template<class, size_t, class, class> class MyVectorSum;

template<class T, size_t N> class MyVector {
    T data[N];
public:
    MyVector<T,N>& operator=(const MyVector<T,N>& right) {
        for(size_t i = 0; i < N; ++i)
            data[i] = right.data[i];
        return *this;
    }
    template<class Left, class Right> MyVector<T,N>&
    operator=(const MyVectorSum<T,N,Left,Right>& right);
    const T& operator[](size_t i) const {
        return data[i];
    }
    T& operator[](size_t i) {
        return data[i];
    }
};

// Allows mixing MyVector and MyVectorSum
template<class T, size_t N, class Left, class Right>
class MyVectorSum {
    const Left& left;
    const Right& right;
public:
    MyVectorSum(const Left& lhs, const Right& rhs)
        : left(lhs), right(rhs) {}
    T operator[](size_t i) const {
        return left[i] + right[i];
    }
};

template<class T, size_t N>
template<class Left, class Right>
MyVector<T,N>&
MyVector<T,N>::
operator=(const MyVectorSum<T,N,Left,Right>& right) {
    for(size_t i = 0; i < N; ++i)
        data[i] = right[i];
    return *this;
}

// operator+ just stores references
template<class T, size_t N>
inline MyVectorSum<T,N,MyVector<T,N>,MyVector<T,N> >
```



```

operator+(const MyVector<T,N>& left,
          const MyVector<T,N>& right) {
    return MyVectorSum<T,N,MyVector<T,N>,MyVector<T,N> >
        (left,right);
}

template<class T, size_t N, class Left, class Right>
inline MyVectorSum<T, N, MyVectorSum<T,N,Left,Right>,
    MyVector<T,N> >
operator+(const MyVectorSum<T,N,Left,Right>& left,
          const MyVector<T,N>& right) {
    return MyVectorSum<T,N,MyVectorSum<T,N,Left,Right>,
        MyVector<T,N> >
        (left, right);
}
// Convenience functions for the test program below
template<class T, size_t N> void init(MyVector<T,N>& v) {
    for(size_t i = 0; i < N; ++i)
        v[i] = rand() % 100;
}

template<class T, size_t N> void print(MyVector<T,N>& v) {
    for(size_t i = 0; i < N; ++i)
        cout << v[i] << ' ';
    cout << endl;
}

int main() {
    srand(time(0));
    MyVector<int, 5> v1;
    init(v1);
    print(v1);
    MyVector<int, 5> v2;
    init(v2);
    print(v2);
    MyVector<int, 5> v3;
    v3 = v1 + v2;
    print(v3);
    // Now supported:
    MyVector<int, 5> v4;
    v4 = v1 + v2 + v3;
    print(v4);
    MyVector<int, 5> v5;
    v5 = v1 + v2 + v3 + v4;
    print(v5);
} ///:~

```

使用模板参数**Left**和**Right**，这个模板很容易地引出一个和的参数类型，来代替上例中指派的那些类型。由于**MyVectorSum**模板持有这额外的两个参数，因此它能表示由**MyVector**和**MyVectorSum**任意组成的一对参数的和。

赋值操作符现在是一个成员函数模板。这将允许任一对**<T,N>**与任一对**<Left,Right>**结合，因此一个**MyVector**对象能够得到来自一个**MyVectorSum**对象的赋值，该**MyVectorSum**对象持有**MyVector**和**MyVectorSum**类型的引用，这两个类型的引用可以组成任何可能的一对。

与前面一样，可以通过跟踪一个简单的赋值操作来准确地了解这个地方发生了些什么事情，从下述表达式开始

```
v4 = v1 + v2 + v3;
```



由于结果表达式变得笨拙冗长且难于处理，在下面的解释中，我们用**MVS**作为**MyVectorSum**的缩写，并且忽略模板的参数。

第1个操作是**v1+v2**，这将调用内联函数**operator+( )**，这个内联函数依次将**MVS(v1, v2)**插入到编译流中。然后它被相加到**v3**上，从而表达式**MVS(MVS(v1, v2), v3)**产生一个临时对象。这个完整语句的最后表达结果是：

```
v4.operator+(MVS(MVS(v1, v2), v3));
```

这种转换完全由编译器安排，它也解释了为什么把这种技术冠以“表达式模板”之名。**MyVectorSum**模板代表了一个表达式（上例中是加法表达式），上述的嵌套调用可能也使读者回忆起左关联表达式**v1+v2+v3**的语法分析树。

由Angelika Langer和Klaus Kreft写的一篇文章解释了这项技术如何扩展到更复杂的计算。<sup>①</sup>

## 5.7 模板编译模型

读者可能已经注意到，所有列举的模板例子都是将完整定义的模板放在每个编译单元中。（例如，将它们完全放在单文件程序中，或者放在多文件程序的头文件中）。这种方法与传统的编程方法背道而驰，传统的编程方法通过将函数声明放在靠后的头文件中，而将函数实现放在独立的文件中（即，**.cpp**）的方法，使得普通函数的定义与它们的声明相分离。

与这种传统方法分离的理由如下：

- 头文件中的非内联函数体会导致多函数的定义，从而导致链接错误。
- 隐藏来自客户有益的函数实现，从而减少了编译时连接。
- 商家可能将预编译代码（为一个特定的编译器编写）分配到各个头文件中，从而使得用户看不到函数的具体实现。
- 头文件越小，编译时间就越短。

### 5.7.1 包含模型

另一方面，模板本质上不是代码，而是产生代码的指令。只有模板的实例化才是真正的代码。当一个编译器在编译期间已经看到了一个完整的模板定义，又在同一个翻译单元内碰到了这个模板实例化点的时候，它就必须涉及这样一个事实：一个相同的实例化点可能会呈现在另一个翻译单元内。处理这种情况最普遍的方法，是在每一个翻译单元内都为这个实例化生成代码，让连接器清除这些副本。另一种特殊的方法也可以很好地处理这种情况，就是用不能被内联的内联函数和虚函数表，这也是为什么虚函数表如此流行的原因之一。但是，有一些编译器宁愿依靠更复杂的机制也不愿意多次产生同一个特定的实例化。C++翻译系统也有责任避免这种由于多个相同的实例化点而产生的错误。

这种方法的一个缺点是，所有的模板源代码对客户都是可见的，因此对于销售库的商家而言，几乎没有机会隐藏他们的实现策略。包含模型的另一个缺点是，头文件比函数体分开编译时大多了。这种方式相比传统编译模型而言，大大增加了编译时间。

为了帮助减少包含模型所需要的大的头文件，C++提供了两个（不惟一的）可供选择的代码组织机制：可以使用显式实例化（explicit instantiation）来手工地实例化每一个模板特化，也可以使用导出模板（exported templates），它支持最大限度的独立的编译。

① Langer和Kreft的文章“C++ Expression Templates”见2003年3月的“C/C++ Users Journal”。也可以参见2003年6月同一本杂志上的由Thomas Becker发表的有关表达式模板的文章（该文章是本节内容素材的来源）。

### 5.7.2 显式实例化

编程人员可以用手工指引编译器实例化他所选择的任何模板特化。当使用这个技术时，对于每个这样的特化，必须有且仅有一条相应的指令；否则可能会收到一个多定义的错误，就像在普通的非内联函数中使用了相同的标识符一样。为了进行示例说明，首先（错误地）将本章前面的例子中的`min()`模板的声明与定义相分离，遵循普通的非内联函数的标准模式。下面的例子包含了5个文件：

- **OurMin.h**：包含`min()`函数模板的声明。
- **OurMin.cpp**：包含`min()`函数模板的定义。
- **UseMin1.cpp**：尝试使用`min()`的一个`int`型实例化。
- **UseMin2.cpp**：尝试使用`min()`的一个`double`型实例化。
- **MinMain.cpp**：调用`usemin1()`和`usemin2()`。

```
//: C05:OurMin.h
#ifndef OURMIN_H
#define OURMIN_H
// The declaration of min()
template<typename T> const T& min(const T&, const T&);
#endif // OURMIN_H ///:~

// OurMin.cpp
#include "OurMin.h"
// The definition of min()
template<typename T> const T& min(const T& a, const T& b) {
    return (a < b) ? a : b;
}

//: C05:UseMin1.cpp {0}
#include <iostream>
#include "OurMin.h"
void usemin1() {
    std::cout << min(1,2) << std::endl;
} ///:~

//: C05:UseMin2.cpp {0}
#include <iostream>
#include "OurMin.h"
void usemin2() {
    std::cout << min(3.1,4.2) << std::endl;
} ///:~

//: C05:MinMain.cpp
//{L} UseMin1 UseMin2 MinInstances
void usemin1();
void usemin2();

int main() {
    usemin1();
    usemin2();
} ///:~
```

当尝试建立这个程序时，连接器报告有未解析的`min<int>()`和`min<double>()`的外部引用。原因是当编译器在`UseMin1`和`UseMin2`中碰到对`min()`特化的调用时，只有`min()`的声明是可见的。由于它的定义不可用，编译器认为它可能来源于某些其他的翻译单元，这样一来在这一点上所需的特化就没有被实例化，从而将问题留给了连接器，连接器最终解释它无法找到它们。

为了解决程序中的这个问题，将引入一个新文件**MinInstances.cpp**，它显式地实例化了所需的**min()**特化：

```
//: C05:MinInstances.cpp {0}
#include "OurMin.cpp"

// Explicit Instantiations for int and double
template const int& min<int>(const int&, const int&);
template const double& min<double>(const double&,
                                   const double&);

///  
~
```

为了手工实例化一个特定的模板特化，可以在该特化的声明前使用**template**关键字。注意，在这里必须包含**OurMin.cpp**而不是**OurMin.h**，这是因为编译器需要用模板定义来进行实例化。然而，这里也是程序中惟一放置该模板定义的地方<sup>①</sup>，因为它提供了程序所需要的独一无二的**min()**的实例化——在其他的文件中只要有其声明就足够了。由于使用了宏预处理器来包含**OurMin.cpp**，因而需要加入包含警告：

```
//: C05:OurMin.cpp {0}
#ifndef OURMIN_CPP
#define OURMIN_CPP
#include "OurMin.h"

template<typename T> const T& min(const T& a, const T& b) {
    return (a < b) ? a : b;
}
#endif // OURMIN_CPP ///  
~
```

现在，当把所有的文件一起编译为一个完整的程序时，就会找到**min()**的惟一的实例，程序就可以正确运行，输出结果如下：

```
1
3.1
```

编程人员也可以手工实例化类和静态数据成员。当显式实例化一个类时，除了一些之前可能已经显式实例化了的成员外，特化所需要的所有成员函数都要进行实例化。这一点很重要，因为使用这种机制时，它必须舍弃很多无用的模板——某些特定的模板将依据它们的参数类型实现不同的功能。隐式实例化在此处有优势：其中只有被调用的成员函数才进行实例化。

显式实例化多用于大型软件工程项目中，因为这样可以避免大量的编译时间。采用隐式实例化还是采用显式实例化完全独立于使用哪一个模板进行编译。可以通过使用包含模型或者分离模型（下节讨论）中的任何一种模型来进行手工实例化。

### 5.7.3 分离模型

模板编译的分离模型跨越了所有的翻译单元，将函数模板定义或者静态数据成员定义从它们的声明中分离出来，就像使用导出（exporting）模板机制下的普通函数和数据一样。在学习了前两节的内容后，这种说法似乎听起来有点儿奇怪。读者首先就可能会问：如果包含模型使用得很顺手，为什么还要怀疑它呢？原因有两个：有历史原因也有技术原因。

从历史上看，包含模型是第1个经历广泛的商品化使用的模型——所有的C++编译器都支持包含模型。其中的部分原因是，在进行标准化的过程中直到该过程后期也没有能够很好地说明

① 正如前面解释的那样，在每一个程序中只能一次显式实例化一个模板。

分离模型，再一个原因就是由于包含模型本身更容易实现一些。在分离模型的语义定下来之前的很长一段时间里，就已经存在很多正在运行着的与之相关的代码了。

分离模型实现起来是如此的困难，以至于直到2003年夏天，仅有一个编译器前端（EDG）支持分离模型。那时，这个编译器如有请求，它仍旧要求模板源代码在编译时可以用来执行实例化操作。方法是在适当的位置使用一些中介代码，来取代总是要求最初的源代码随时准备好以备使用的形式。这样就可以在不需要传递源代码的情况下传递某些“预编译”模板。鉴于本章前面介绍过的查找的复杂性（就是有关在模板定义的语境中查找关联名称的内容），当编译一个实例化某个模板的程序时，仍然要以某种形式来使用一个完整模板的定义。

将一个模板定义的源代码与它的声明相分离的程序语法是很简单的。只要使用**export**关键字就可以了：

```
// C05:OurMin2.h
// Declares min as an exported template
// (Only works with EDG-based compilers)
#ifndef OURMIN2_H
#define OURMIN2_H
export template<typename T> const T& min(const T&,
                                         const T&);
#endif // OURMIN2_H ///:~
```

类似于**inline**或者**virtual**，关键字**export**在一个编译流中仅需出现一次。在这个编译流中，引入了一个导出模板。由于这个原因，我们在实现文件中无需重复它，但是再对它进行一下声明是一个好习惯。

```
// C05:OurMin2.cpp
// The definition of the exported min template
// (Only works with EDG-based compilers)
#include "OurMin2.h"
export
template<typename T> const T& min(const T& a, const T& b) {
    return (a < b) ? a : b;
} ///:~
```

之前用过的**UseMin**文件只需包含正确的头文件（**OurMin2.h**），主程序不用改动。尽管看起来这已经产生了正确的分离，但带有模板定义的文件（**OurMin2.cpp**）仍然必须传递给用户（因为**min()**的每一个实例化都必须进行处理），直至遇到这样的情况：表示模板定义的某种中介代码形式得到支持。因此，当一个正确的分离模型标准提出来的时候，其中所有的好处并不会都在今天马上体现出来。当今只有一个编译器家族支持**export**（那些基于EDG前端的编译器），而且这些编译器当前并没有开发将模板定义分配到已编译的格式中的潜力。

## 5.8 小结

模板广泛使用的程度远远超过简单的类型参数化。当对模板结合使用参数类型推断、用户自定义特化和模板元编程的时候，C++模板作为一种强有力的代码产生机制已经形成了。

这里有一个我们没有提及的C++模板的缺陷，就是在解释编译时错误信息报告方面的困难。由于编译器产生总量难以预料的出错信息报告文本可能是完全无法避免的。现在，C++编译器已经改进了它们的模板错误信息报告方式，此外Leor Zolman也已开发了一种名为**STLFilt**的工具，这种工具采用提取有用信息和抛掉冗余信息<sup>①</sup>的方式，使得汇报的这些错误信息更具可读性。

① 访问<http://www.bdsoft.com/tools/stlfilt.html>。

读者从本章得到的另一个重要的思想就是，一个模板意味着一个接口。也就是说，尽管关键字**template**意味着：“我可以接受任何类型的参数”，但在模板定义中的代码也要求某些需要提供支持的运算符和成员函数——这些运算符和成员函数就是接口。因此，实际上一个模板定义意味着：“我将接受任何支持这个接口的参数”。若编译器能够仅仅说：“嗨，这种实例化模板的类型参数不支持这个接口——不能使用这个类型”，事情会变得更好一些。运用模板构成的带有“隐含的类型检查”的接口机制，比起那些要求所有的类型都必须从某些基类派生出来的纯面向对象的使用惯例来说更加灵活。

在第6章和第7章中，将深入探讨模板最著名的应用：标准C++库的子集，即广为人知的标准模板库（STL）。第9章和第10章中也用到了本章未提及的某些模板技术。

## 5.9 练习

- 5-1 编写一个具有单一类型模板参数的一元函数模板。用类型**int**生成它的一个完全特化。再为这个拥有单一的**int**参数的函数产生一个非模板重载。在主函数中调用这三个函数。
- 5-2 编写一个类模板，该类模板用**vector**实现一个栈数据结构。
- 5-3 对习题（5-2）的解答进行修改，使得用来实现栈的容器的类型是一个模板类型的模板参数。
- 5-4 在下面的代码中，类**NonComparable**中没有**operator=()**。解释一下为什么类**HardLogic**的出现引起了一个编译错误，而**SoftLogic**却没有？

```

//: C05:Exercise4.cpp {-xo}
class Noncomparable {};

struct HardLogic {
    Noncomparable nc1, nc2;
    void compare() {
        return nc1 == nc2; // Compiler error
    }
};

template<class T> struct SoftLogic {
    Noncomparable nc1, nc2;
    void noOp() {}
    void compare() {
        nc1 == nc2;
    }
};

int main() {
    SoftLogic<Noncomparable> l;
    l.noOp();
} //:~

```

- 5-5 编写一个持有单一类型参数（**T**）的函数模板，它接受了4个函数参数：一个**T**类数组、一个开始索引值、一个结束索引值（在允许范围之内的）和一个可选的初始值。函数返回指定范围内所有数组元素值与初始值的和。用默认构造函数为**T**类型的数据用默认方式赋初值。
- 5-6 重做上面的习题，根据本章讨论过的技术，使用显式实例化来手工生成**int**和**double**的特化。
- 5-7 请指出为什么下列代码无法编译？（提示：看看类成员函数访问了什么？）

```

//: C05:Exercise7.cpp {-xo}
class Buddy {};

template<class T> class My {
    int i;
public:
    void play(My<Buddy>& s) {
        s.i = 3;
    }
};

int main() {
    My<int> h;
    My<Buddy> me, bud;
    h.play(bud);
    me.play(bud);
} ///:~

```

5-8 指出下面的代码为什么无法编译?

```

//: C05:Exercise8.cpp {-xo}
template<class T> double pythag(T a, T b, T c) {
    return (-b + sqrt(double(b*b - 4*a*c))) / 2*a;
}

int main() {
    pythag(1, 2, 3);
    pythag(1.0, 2.0, 3.0);
    pythag(1, 2.0, 3.0);
    pythag<double>(1, 2.0, 3.0);
} ///:~

```

- 5-9 编写一些持有下列多种无类型参数的模板：一个**int**、一个指向**int**的指针、一个指向**int**类型的静态类成员的指针和一个指向一个静态成员函数的指针。
- 5-10 编写一个持有两个类型参数的类模板。为第1个参数定义半特化，另一个半特化指定第2个参数。在每一个特化中，都采用没有出现在基本模板中的成员。
- 5-11 定义一个持有单一类型参数的类模板，将其命名为**Bob**。使**Bob**成为一个名为**Friendly**的模板类的所有实例的友元，并且成为一个名为**Picky**的类模板的友元——仅当**Bob**和**Picky**的类型参数完全相同的时候。提供一些能证明这些类的友元关系的**Bob**成员函数。



## 通用算法

算法是计算的核心。能够编写出在任何一种类型序列下工作的算法，就可以使程序更加简单和安全。算法在运行时的自定义的能力革新了软件开发方式。

众所周知，标准模板库（Standard Template Library, STL）作为标准C++库的子集，最初是用来设计通用算法（generic algorithm）的——以类型安全的方式生成处理任何一种类型值序列的代码。以前每当需要处理一个数据集合的时候，就得重复地手工编写代码。设计STL的目标就是对于几乎每个任务都使用预定义的算法，来代替这种手工编码的工作。然而，这种方法在提供方便的同时，也为初学者的学习带来了一些困难。在学习完本章后，读者就能自己做出判定，是特别喜欢采用这些算法来进行编程还是越学越困惑。大多数人起初抵制算法的使用，但随着时间的推移，越来越多的人逐渐喜欢使用它们了。

### 6.1 概述

除了一些别的东西，标准库中的通用算法还提供一个词汇表来描述各种解法。随着对算法的熟悉，读者就会获得一个新的词汇集合用来讨论现在正在做什么，这些词汇往往比以前所用的词汇具有更高层次的抽象。例如，没有必要说“这个循环在运行过程中从这赋值到那，……，噢，我知道了，是复制！”，而是只需简单扼要地用**copy()**就可以了。这就是初学者在早期的计算机编程中所做的——创建高层次的抽象来解释正在做什么以及用更少的时间来说明怎样去做。一旦解决了怎样做的问题，并且将其转换成代码隐藏于算法代码中，就可以在需要时重复使用这些算法。

这里有一个怎样使用**copy**算法的程序例子：

```
//: C06:CopyInts.cpp
// Copies ints without an explicit loop.
#include <algorithm>
#include <cassert>
#include <cstdint> // For size_t
using namespace std;

int main() {
    int a[] = { 10, 20, 30 };
    const size_t SIZE = sizeof a / sizeof a[0];
    int b[SIZE];
    copy(a, a + SIZE, b);
    for(size_t i = 0; i < SIZE; ++i)
        assert(a[i] == b[i]);
} ///:~
```

**copy()**算法的前两个参数表示输入序列的范围——此处是数组**a**。范围用一对指针表示。第1个指针指向该序列的第1个元素，第2个指针指向数组的超越末尾的（past the end）位置（即数组的最后一个元素的后面）。刚开始看起来可能觉得比较陌生，但这是传统的C语言的习惯用法，可以带来很大的便利。例如，这两个指针的差值就是序列中元素的个数。更重要的是，在实现**copy**时，第2个指针可以作为在序列中停止迭代的标记符。第3个参数代表输出序列的开始位置，在本例中输出序列是数组**b**。这里假设**b**表示的数组有足够的空间来接收要复制的

元素。

如果**copy()**算法仅限于处理整数，那就没什么新奇的地方了。它还可以用于任何一种类型的序列。下面的例子用来复制**string**对象：

```
//: C06:CopyStrings.cpp
// Copies strings.
#include <algorithm>
#include <cassert>
#include <cstdint>
#include <string>
using namespace std;

int main() {
    string a[] = {"read", "my", "lips"};
    const size_t SIZE = sizeof a / sizeof a[0];
    string b[SIZE];
    copy(a, a + SIZE, b);
    assert(equal(a, a + SIZE, b));
} ///:~
```

这个例子引入了另一个算法**equal()**，仅当第1个序列的每一个元素与第2个序列的对应元素相等时（使用**operator==( )**）返回**true**。这个例子对每个序列遍历了两次，一次用来复制，一次用来比较，而不是采用单一的一次循环！

通用算法能够达到如此灵活性是由于采用了函数模板。如果读者能将**copy()**的实现想象成下面形式，这样差不多就对了：

```
template<typename T> void copy(T* begin, T* end, T* dest) {
    while(begin != end)
        *dest++ = *begin++;
}
```

说“差不多”是因为**copy()**能够处理这样一类的序列，该序列由类似指针的任意类型来限制，例如迭代器。以此方式，**copy()**可以用来复制**vector**：

```
//: C06:CopyVector.cpp
// Copies the contents of a vector.
#include <algorithm>
#include <cassert>
#include <cstdint>
#include <vector>
using namespace std;

int main() {
    int a[] = { 10, 20, 30 };
    const size_t SIZE = sizeof a / sizeof a[0];
    vector<int> v1(a, a + SIZE);
    vector<int> v2(SIZE);
    copy(v1.begin(), v1.end(), v2.begin());
    assert(equal(v1.begin(), v1.end(), v2.begin()));
} ///:~
```

第1个**vector**对象**v1**由数组**a**中的整数序列来初始化。第2个**vector**对象**v2**的定义，使用一个不同的能够为**SIZE**个元素分配空间的**vector**构造函数，并且将其初始化为0（整型的默认值）。

如同前面的数组例子，**v2**要有足够的空间来接收**v1**内容的复制。为了方便起见，使用一个特殊的库函数**back\_inserter()**，该函数返回一个特殊类型的迭代器。利用这个迭代器就可以用插入元素的方式来代替重写这些元素，因此内存的大小就可以根据容器的需要自动扩大。



下面的例子使用了**back\_inserter()**，因此无需像前面例子那样，在建立输出**vector**的对象**v2**时必须确定其大小。

```

//: C06:InsertVector.cpp
// Appends the contents of a vector to another.
#include <algorithm>
#include <cassert>
#include <cstdint>
#include <iterator>
#include <vector>
using namespace std;

int main() {
    int a[] = { 10, 20, 30 };
    const size_t SIZE = sizeof a / sizeof a[0];
    vector<int> v1(a, a + SIZE);
    vector<int> v2; // v2 is empty here
    copy(v1.begin(), v1.end(), back_inserter(v2));
    assert(equal(v1.begin(), v1.end(), v2.begin()));
} //:~

```

**back\_inserter()**函数在头文件**<iterator>**中定义。我们将在下一章详细解释插入迭代器是如何工作的。

迭代器在本质上与指针相同，所以可以在标准库中以一种能够接受迭代器和指针两种参数的方式来实现算法。因此，**copy()**的实现如下所示：

```

template<typename Iterator>
void copy(Iterator begin, Iterator end, Iterator dest) {
    while(begin != end)
        *begin++ = *dest++;
}

```

调用时无论采用哪种类型的参数，**copy()**都假定它正确地实现了间接引用和自增运算符。如果没有，就会得到一个编译时错误。

### 6.1.1 判定函数

有时只想复制定义好的某个序列中的一个子集到另一个序列中，这个子集由只满足某个特殊条件的那些元素组成。为了达到这种灵活性，很多算法的调用序列允许提供一个判定函数(predicate)，即一个基于某种标准返回布尔型值的函数。例如，只想提取整数序列中那些小于或等于15的数。**copy()**的一种称为**remove\_copy\_if()**的版本能够完成这一工作，如下所示：

```

//: C06:CopyInts2.cpp
// Ignores ints that satisfy a predicate.
#include <algorithm>
#include <cstdint>
#include <iostream>
using namespace std;

// You supply this predicate
bool gt15(int x) { return 15 < x; }

int main() {
    int a[] = { 10, 20, 30 };
    const size_t SIZE = sizeof a / sizeof a[0];
    int b[SIZE];
    int* endb = remove_copy_if(a, a+SIZE, b, gt15);
    int* beginb = b;
    while(beginb != endb)

```



```
    cout << *beginb++ << endl; // Prints 10 only
} ///:~
```

**remove\_copy\_if()** 函数模板需要一些通常用来限定范围的指针，还增加了一个用户自选的判定函数。判定函数必须是指向一个函数<sup>①</sup>的指针，这个指针有一个与序列中元素同类型的参数，并且必须返回一个布尔型的值。在这里，当参数大于15时，函数**gt15**返回**true**。**remove\_copy\_if()** 算法对输入序列的每个元素都应用**gt15()**，并且在向输出序列写入时忽略掉那些使判定函数产生真值的元素。

下面的程序展示了**copy**算法的另外一个变种：

```
///: C06:CopyStrings2.cpp
// Replaces strings that satisfy a predicate.
#include <algorithm>
#include <cstdint>
#include <iostream>
#include <string>
using namespace std;

// The predicate
bool contains_e(const string& s) {
    return s.find('e') != string::npos;
}

int main() {
    string a[] = {"read", "my", "lips"};
    const size_t SIZE = sizeof a / sizeof a[0];
    string b[SIZE];
    string* endb = replace_copy_if(a, a + SIZE, b,
        contains_e, string("kiss"));
    string* beginb = b;
    while(beginb != endb)
        cout << *beginb++ << endl;
} ///:~
```

与刚才忽略不满足判定函数的元素不同，此例中的**replace\_copy\_if()**在输出一个序列时用一个固定的值来替代这些元素。输出结果是：

```
kiss
my
lips
```

因为“read”是几个输入字符串中惟一个含有字母e的字符串，所以用字符串“kiss”来取代该字符串，“kiss”是**replace\_copy\_if()**调用中指定的最后一个参数。

**replace\_if()** 算法改变原始序列相应位置中的内容，而不是向单独的输出序列中写数据，程序如下所示：

```
///: C06:ReplaceStrings.cpp
// Replaces strings in-place.
#include <algorithm>
#include <cstdint>
#include <iostream>
#include <string>
using namespace std;

bool contains_e(const string& s) {
```

① 或者是和函数一样可被调用的东西，我们将很快能遇到。

```

    return s.find('e') != string::npos;
}

int main() {
    string a[] = {"read", "my", "lips"};
    const size_t SIZE = sizeof a / sizeof a[0];
    replace_if(a, a + SIZE, contains_e, string("kiss"));
    string* p = a;
    while(p != a + SIZE)
        cout << *p++ << endl;
} ///:~

```

### 6.1.2 流迭代器

像任何好的软件库一样，标准C++库试图提供便捷的方法以自动完成常见的任务。在本章开始部分曾经提到过，可以使用通用算法来取代循环结构的设想。但是到目前为止，在提出的例子中仍然直接使用循环来打印输出结果。因为打印输出结果是最常见的任务之一，也可以期望有一种方法能够自动实现它。

这里引入流迭代器（stream iterator）的概念。一个流迭代器使用流作为输入或输出序列。例如，为了去除程序**CopyInts2.cpp**中的输出循环，可以像下面这样做：

```

//: C06:CopyInts3.cpp
// Uses an output stream iterator.
#include <algorithm>
#include <cstdint>
#include <iostream>
#include <iterator>
using namespace std;

bool gt15(int x) { return 15 < x; }

int main() {
    int a[] = { 10, 20, 30 };
    const size_t SIZE = sizeof a / sizeof a[0];
    remove_copy_if(a, a + SIZE,
                   ostream_iterator<int>(cout, "\n"), gt15);
} ///:~

```

在本例中，**remove\_copy\_if()**的第3个参数位置上的输出序列**b**用一个输出流迭代器来代替，这个迭代器是在头文件**<iterator>**中声明的**ostream\_iterator**类模板的一个实例。输出流迭代器重载其拷贝-赋值操作符，该重载操作符向相应的流写数据。**ostream\_iterator**的这个特殊实例应用于输出流**cout**。每次**remove\_copy\_if()**通过迭代器对**cout**赋一个来自输入序列**a**的整数。即迭代器向**cout**写入这个整数，并且随后还自动写入一个单独的字符串的一个实例，该字符串位于它的第2个参数位置上，在本例中是一个换行符。

用一个输出文件流来代替**cout**，就使得写文件同样很容易实现：

```

//: C06:CopyIntsToFile.cpp
// Uses an output file stream iterator.
#include <algorithm>
#include <cstdint>
#include <fstream>
#include <iterator>
using namespace std;

bool gt15(int x) { return 15 < x; }

int main() {
    int a[] = { 10, 20, 30 };

```

```

const size_t SIZE = sizeof a / sizeof a[0];
ofstream outf("ints.out");
remove_copy_if(a, a + SIZE,
               ostream_iterator<int>(outf, "\n"), gt15);
} ///:~

```

一个输入流迭代器允许算法从输入流中获得它的输入序列。这是靠构造函数和 **operator++()** 从基础的流中读入下一个元素，以及重载 **operator\*()** 产生先前读入的值来完成的。因为算法需要两个指针来限定输入序列，所以可以用两种方式构造 **istream\_iterator**，请看下面的程序：

```

//: C06:CopyIntsFromFile.cpp
// Uses an input stream iterator.
#include <algorithm>
#include <fstream>
#include <iostream>
#include <iterator>
#include "../require.h"
using namespace std;

bool gt15(int x) { return 15 < x; }

int main() {
    ofstream ints("someInts.dat");
    ints << "1 3 47 5 84 9";
    ints.close();
    ifstream inf("someInts.dat");
    assure(inf, "someInts.dat");
    remove_copy_if(istream_iterator<int>(inf),
                  istream_iterator<int>(),
                  ostream_iterator<int>(cout, "\n"), gt15);
} ///:~

```

程序中 **replace\_copy\_if()** 的第1个参数，把一个 **istream\_iterator** 的对象应用于含有整数的输入文件流。第2个参数使用 **istream\_iterator** 类的默认构造函数。这个调用构造了 **istream\_iterator** 的一个特殊值，用以指示文件的结束。这样，当第1个迭代器最终遇到物理文件的结尾时，它与 **istream\_iterator<int>()** 的值进行是否相等的比较，以便算法正确结束。注意，本例中完全避免了直接使用数组。

### 6.1.3 算法复杂性

使用某个软件库是对它的一种信任。用户相信（软件库的）实现者不仅能提供正确的功能，并且希望这些功能能够尽可能有效地执行。与其使用性能低下的算法，还不如自己来编写循环代码。

为了保证库实现的质量，C++标准不仅说明了算法应该做什么，而且说明了包括做得有多快，有时还包括应该使用多少存储空间。不能满足性能需求的算法也就是不符合标准的算法。算法的执行效率的度量称为复杂性（complexity）。

如果可能的话，C++标准会指定一个算法应该耗费的操作的精确次数。例如，**count\_if()** 算法返回一个序列中满足给定判定函数的元素的个数。下面对 **count\_if()** 的调用，如果应用于类似本章前面的例子中的整数序列，会产生大于15的整数元素的个数：

```
size_t n = count_if(a, a + SIZE, gt15);
```

因为 **count\_if()** 必须对每个元素仔细检查一次，也就是比较的次数与序列中元素的个数肯定相等。**copy()** 算法有相同的规格说明。

对于其他算法可以指定其最多执行的操作次数。**find()**算法要搜索一个序列，直到遇到一个等于它的第3个参数的元素：

```
int* p = find(a, a + SIZE, 20);
```

只要找到这样的元素就停止查找，并且返回一个指针，这个指针指向该元素第1次出现的位置。如果一个也没有找到，也返回一个指针，该指针指向超越序列末尾的（本例中是 **a+SIZE**）位置。所以，**find()**比较的次数最多等于序列中元素的个数。

有时候不能精确衡量一个算法将耗费运算的次数。在这种情况下，C++标准给出算法的渐近复杂性(asymptotic complexity)，这是对算法在大的序列输入下执行性能与已知公式相比较的度量。一个好的例子是**sort()**算法，C++标准称其花费“在平均情况下约 $n \log n$ 次比较”（ $n$ 是序列中元素的个数）<sup>①</sup>。这样的复杂性度量似乎给人们一种关于一个算法的开销的“感觉”，不管怎么样，这至少是一种用来比较算法性能的有意义的依据。在下一章中读者将会看到，对于容器**set**的成员函数**find()**来说，它具有对数级的复杂性，这意味着在大的集合中查找所花费的时间与元素个数的对数成正比。这比元素个数 $n$ 要小的多，因此查找一个**set**最好使用它自己的成员函数**find()**而不用一般的**find()**算法。

## 6.2 函数对象

学习本章前面的例子，读者可能会注意到函数**gt15()**的使用限制。如果用其他数而不是15来作为比较的阈值该怎么办？可能会需要**gt20()**或**gt25()**等等。为它们再编写单独的函数会耗费时间而且不合理，因为当编写应用代码时程序员需要知道所有要求的值。

后者的限制意味着不能使用运行时的值<sup>②</sup>来控制查找，这是不能接受的。为了克服这个困难，需要有一种在运行时把信息传递给判定函数的方式。例如，程序员可能需要一个能用任意比较值来初始化一个判定大于的函数（greater-than function）。遗憾的是，不能把这个值作为一个函数参数进行传递，因为这是个一元判定函数，比如**gt15()**。它单独地应用于序列中的每一个值，因此必须只能有一个参数。

和往常一样，跳出这个两难的局面的方法就是创建一个抽象。在这里，需要这样的一个抽象，它实现起来像函数同时保存状态，使用时却不用考虑函数参数的个数。这种抽象称为函数对象(function object)。<sup>③</sup>

函数对象是重载了**operator()**的类的一个实例，**operator()**是函数调用运算符。这个运算符允许用函数调用语法并使用对象。如同其他对象一样，可以通过该对象的构造函数来初始化它。下面程序中的函数对象可以取代**gt15()**：

```
//: C06:GreaterThanN.cpp
#include <iostream>
using namespace std;

class gt_n {
    int value;
public:
    gt_n(int val) : value(val) {}
    bool operator()(int n) { return n > value; }
};
```

① 这是 $O(n \log n)$ 的英文简单表述，其表示对于大的 $n$ 比较的次数与函数 $f(n) = n \log n$ 成正比。

② 除非使用全局变量来烦琐地实现。

③ 函数对象也称函数子(functor)，函数子是一个具有类似行为的数学概念。

```
int main() {
    gt_n f(4);
    cout << f(3) << endl; // Prints 0 (for false)
    cout << f(5) << endl; // Prints 1 (for true)
} ///:~
```

当创建函数对象**f**时，传递作为比较对照的固定值（4）。编译器像下面的函数调用一样计算表达式**f(3)**：

```
f.operator()(3);
```

返回值是表达式**3>value**的值，在本例中当**value**是4时为假。

因为这样的比较还可以应用于除**int**外的其他类型，只要把**gt\_n()**定义为一个类模板就有意义了。无需用户亲自做——标准库已经帮你做了。下面对函数对象的描述不仅使这个主题更清晰，而且读者对通用算法如何工作有了一个更好的理解。

### 6.2.1 函数对象的分类

标准C++库根据函数对象的运算符**operator()**使用参数的个数和返回值的类型对其进行分类。这种分类是基于函数对象的运算符**operator()**使用参数的个数分别为零个、一个或两个的情况进行：

**发生器(Generator)**：一种没有参数且返回一个任意类型值的函数对象。随机数发生器就是发生器的一个例子。标准库提供一个发生器，就是在**<cstdlib>**中声明的函数**rand()**以及一些算法，如**generate\_n()**，它将发生器应用于序列。

**一元函数(Unary Function)**：一种只有一个任意类型的参数，且返回一个可能不同类型（比如可能是**void**）值的函数对象。

**二元函数(Binary Function)**：一种有两个任意类型的（可能是不同类型）参数，且返回一个任意类型（包括**void**）值的函数对象。

**一元判定函数(Unary Predicate)**：返回**bool**型值的一元函数。

**二元判定函数(Binary Predicate)**：返回**bool**型值的二元函数。

**严格弱序(Strict Weak Ordering)**：一种更广义地理解“相等”概念的二元判定函数。一些标准容器认为在两个元素中，如果任何一个都不小于另外一个（使用**operator<()**），则二者相等。这对于比较浮点型值及其他类型的对象很重要，因为**operator==( )**是不可靠的或者说是不可行的。同时这种概念也适用于想在**struct**的所有字段的一个子集上对数据记录（**struct**）的序列进行排序的情况。这种比较方案被认为是一种严格弱序，因为具有相等关键字集（equal key）的两个记录作为对象的整体而言，两个记录不是真正的“相等”，但对于正在使用的这个比较来说是相等的。这种观念的重要性在下一章中将会更加明显。

另外，某些算法假定对它们处理的对象类型的有关操作是有效的。现在用下面这些术语来介绍这些假定：

**小于可比较(LessThanComparable)**：含有小于运算符**operator<**的类。

**可赋值(Assignable)**：含有对于同类型指定赋值操作符**operator=**的类。

**相等可比较(EqualityComparable)**：含有对于同类型相等运算符**operator==**的类。

在本章后面将使用这些术语来描述标准库中的通用算法。

### 6.2.2 自动创建函数对象

头文件**<functional>**定义了大量有用的通用函数对象。毋庸置疑，它们是很简单的，但可以用它们来组成更加复杂的函数对象。因此，在大多数情况下，无需编写任何函数就可以构

造出复杂的判定函数。可以用函数对象适配器 (function object adaptor)<sup>⑨</sup> 来获得一个简单的函数对象, 并且调整它们用来与操作链中的其他函数对象配合。

举例说明, 现在仅使用标准函数对象来完成前面介绍的**gt15()**的工作。标准函数对象**greater**是一个二元函数对象, 当它的第1个参数大于第2个参数时返回**true**。不能通过一个算法如**remove\_copy\_if()**直接把它应用到整数序列, 因为**remove\_copy\_if()**是一个一元判定函数。可以通过用**greater**的第1个参数与某个固定值进行比较的方式, 来构造一个一元判定函数。用函数对象适配器**bind2nd**作为固定的第2个参数的值, 其值为15, 如下列程序所示:

```
//: C06:CopyInts4.cpp
// Uses a standard function object and adaptor.
#include <algorithm>
#include <cstdint>
#include <functional>
#include <iostream>
#include <iterator>
using namespace std;

int main() {
    int a[] = { 10, 20, 30 };
    const size_t SIZE = sizeof a / sizeof a[0];
    remove_copy_if(a, a + SIZE,
                   ostream_iterator<int>(cout, "\n"),
                   bind2nd(greater<int>(), 15));
} ///:~
```

这个程序没用前面用户自己定义的判定函数**gt15()**, 却产生了与**CopyInts3.cpp**相同的结果。函数对象适配器**bind2nd()**是一个模板函数, 它创建一个**binder2nd**类型的函数对象。仅存储和传递两个参数给**bind2nd()**, 其中第1个参数必须是一个二元函数或函数对象 (即带有两个参数的可以被调用的任意对象)。**binder2nd**中的**operator()**函数, 它本身是一个一元函数, 该函数调用存储的二元函数, 并传递引入的参数及其存储的固定值。

为了更具体地解释这个例子, 现在调用由**bind2nd()**创建的**binder2nd**的一个名为**b**的实例。当创建**b**时, 它接收两个参数 (**greater<int>()**和15) 并且保存它们。调用名为**g**的**greater<int>**的实例和名为**o**的输出流迭代器的实例。这时在前面的程序中对**remove\_copy\_if()**的调用在概念上可表示成下面这样:

```
remove_copy_if(a, a + SIZE, o, b(g, 15).operator());
```

伴随着**remove\_copy\_if()**在序列中的迭代, 对每个元素调用**b**来决定当复制到目的流时是否忽略该元素。如果用**e**来标记当前元素, **remove\_copy\_if()**中的调用等价于:

```
if(b(e))
```

但是**binder2nd**的函数调用运算符还要回来调用**g(e,15)**, 所以上面的调用与下面的调用一样:

```
if(greater<int>(e, 15))
```

这就是我们要寻求的比较。这里还有一个**bind1st()**适配器, 它创建一个**binder1st**对象, 该对象是相关联的输入二元函数确定的第一个参数。

这里有另外一个例子, 用来计算某个序列中不等于20的元素的个数。这次使用前面介绍过

⑨ 依照C++标准, 在这里的写成adaptor。在关于设计模式章节中我们根据习惯使用adapter。这两种写法都是可接受的。

的算法**count\_if()**。程序中有一个标准二元函数对象**equal\_to**，还有一个函数对象适配器**not1()**，该函数对象适配器以一元函数对象作为参数并转化其实际值。下面的程序将会完成这个任务：

```
//: C06:CountNotEqual.cpp
// Count elements not equal to 20.
#include <algorithm>
#include <cstdint>
#include <functional>
#include <iostream>
using namespace std;

int main() {
    int a[] = { 10, 20, 30 };
    const size_t SIZE = sizeof a / sizeof a[0];
    cout << count_if(a, a + SIZE,
                     not1(bind1st(equal_to<int>(), 20))) // 2
    << endl;
}
```

如在前面的例子中的**remove\_copy\_if()**一样，**count\_if()**调用由位于它的第3个参数（称其为**n**）位置的函数对序列中的每一个元素进行判定，并且在每次返回**true**时使其内部的计数器增1。如前所述，如果称序列中当前元素为**e**，则语句

```
if(n(e))
```

在**count\_if**实现过程中，可以解释为

```
if(!bind1st(equal_to<int>, 20)(e))
```

结束时如下所示：

```
if(!equal_to<int>(20, e))
```

这是因为**not1()**返回的是调用它的一元函数参数的结果的逻辑否定。**equal\_to**的第1个参数是20，因为在这里用**bind1st()**来代替**bind2nd()**。由于在参数中相等性测试是对称的，在这个例子中可以使用**bind1st()**或**bind2nd()**。

下面的表格显示了产生标准函数对象的模板，还显示了模板应用的表达式的种类：

名 称	类 型	产生的结果
<b>plus</b>	BinaryFunction	arg1+arg2
<b>minus</b>	BinaryFunction	arg1-arg2
<b>multiplies</b>	BinaryFunction	arg1*arg2
<b>divides</b>	BinaryFunction	arg1/arg2
<b>modulus</b>	BinaryFunction	arg1%arg2
<b>negate</b>	UnaryFunction	-arg1
<b>equal_to</b>	BinaryPredicate	arg1==arg2
<b>not_equal_to</b>	BinaryPredicate	arg1!=arg2
<b>greater</b>	BinaryPredicate	arg1>arg2
<b>less</b>	BinaryPredicate	arg1<arg2
<b>greater_equal</b>	BinaryPredicate	arg1>=arg2
<b>less_equal</b>	BinaryPredicate	arg1<=arg2
<b>logical_and</b>	BinaryPredicate	arg1&&arg2
<b>Logical_or</b>	BinaryPredicate	arg1  arg2
<b>logical_not</b>	UnaryPredicate	!arg1
<b>unary_negate</b>	Unary Logical	!(UnaryPredicate(arg1))
<b>binary_negate</b>	Binary Logical	!(BinaryPredicate(arg1,arg2))



### 6.2.3 可调整的函数对象

标准函数适配器例如**bind1st()**和**bind2nd()**，对它们处理的函数对象做一些相关的假设。考虑前面的**CountNotEqual.cpp**程序中最后一行的表达式：

```
not1(bind1st(equal_to<int>(), 20))
```

**bind1st()**适配器创建了一个**binder1st**类型的一元函数对象，它仅存储**equal\_to<int>**的一个实例及值20。函数**binder1st::operator()**需要知道它的参数类型和它的返回值类型；否则，它就不是一个有效的声明。解决这一问题的简便方式是，期望所有的函数对象提供这些类型的嵌套类型定义。对于一元函数，是类型名为**argument\_type**和**result\_type**；对于二元函数对象，为**first\_argument\_type**、**second\_argument\_type**和**result\_type**。看看头文件**<functional>**中**bind1st()**和**binder1st**的实现就显示了这一期望。首先检查一下可能出现在典型的库实现中的**bind1st()**：

```
template<class Op, class T>
binder1st<Op> bind1st(const Op& f, const T& val) {
    typedef typename Op::first_argument_type Arg1_t;
    return binder1st<Op>(f, Arg1_t(val));
}
```

注意，模板参数**Op**，代表正在由**bind1st()**调整的二元函数的类型，它必须含有一个名为**first\_argument\_type**的嵌套类型。（注意，如同在第5章中解释的那样，使用**typename**来通知编译器它是一个成员类型名。）现在看看**binder1st**在它的函数调用运算符的声明中如何使用**Op**中的类型名：

```
// Inside the implementation for binder1st<Op>
typename Op::result_type
operator()(const typename Op::second_argument_type& x)
    const;
```

为这些类提供类型名的函数对象，称为可调整的函数对象（adaptable function object）。

因为所有的标准函数对象以及用户为了使用函数对象适配器而自己创建的函数对象都期望这些类型名称，所以头文件**<functional>**提供了两种模板来定义这些类型：**unary\_function**和**binary\_function**。当为派生自这些类的简单函数对象填写参数类型时，可以由这些类型作为模板参数。例如，假设要想使本章前面定义的函数对象**gt\_n**成为可调整的，我们需要做的工作如下所示：

```
class gt_n : public unary_function<int, bool> {
    int value;
public:
    gt_n(int val) : value(val) {}
    bool operator()(int n) {
        return n > value;
    }
};
```

所有的**unary\_function**都提供合适的类型定义，这些正如读者在定义中所见到的，由模板参数推断而来：

```
template<class Arg, class Result> struct unary_function {
    typedef Arg argument_type;
    typedef Result result_type;
};
```

这些类型通过**gt\_n**变成可使用的，因为它是从**unary\_function**公有派生来的。

**binary\_function**模板使用起来与其类似。

#### 6.2.4 更多的函数对象例子

下面的**FunctionObjects.cpp**例子，对多数内建的基本函数对象模板提供了简单的测试。读者可以看到，用这种方式如何使用每个模板，并取得相应的操作结果。为简便起见，在这个例子中使用了下面这些发生器当中的一个：

```
//: C06:Generators.h
// Different ways to fill sequences.
#ifndef GENERATORS_H
#define GENERATORS_H
#include <cstring>
#include <set>
#include <cstdlib>

// A generator that can skip over numbers:
class SkipGen {
    int i;
    int skip;
public:
    SkipGen(int start = 0, int skip = 1)
        : i(start), skip(skip) {}
    int operator()() {
        int r = i;
        i += skip;
        return r;
    }
};

// Generate unique random numbers from 0 to mod:
class URandGen {
    std::set<int> used;
    int limit;
public:
    URandGen(int lim) : limit(lim) {}
    int operator()() {
        while(true) {
            int i = int(std::rand()) % limit;
            if(used.find(i) == used.end()) {
                used.insert(i);
                return i;
            }
        }
    }
};

// Produces random characters:
class CharGen {
    static const char* source;
    static const int len;
public:
    char operator()() {
        return source[std::rand() % len];
    }
};
#endif // GENERATORS_H ///:~

//: C06:Generators.cpp {0}
#include "Generators.h"
const char* CharGen::source = "ABCDEFGHIIJK"
    "LMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
```



```
const int CharGen::len = std::strlen(source);
///  
~
```

在本章中的后面部分，将在列举的各种各样的例子中使用这些生成函数。**SkipGen**函数对象，返回一个算术序列当前元素的下一个数，它们共同的差值保存在数据成员**skp**中。**UrandGen**对象在指定范围内产生一个惟一的随机数。（它使用**set**容器，**set**容器将在下一章中介绍。）**CharGen**对象返回一个随机的字母表中的字符。下面是一个使用**UrandGen**的程序例子：

```
///  
C06:FunctionObjects.cpp {-bor}  
// Illustrates selected predefined function object  
// templates from the Standard C++ library.  
//{L} Generators  
#include <algorithm>  
#include <cstdlib>  
#include <ctime>  
#include <functional>  
#include <iostream>  
#include <iterator>  
#include <vector>  
#include "Generators.h"  
#include "PrintSequence.h"  
using namespace std;  
  
template<typename Contain, typename UnaryFunc>  
void testUnary(Contain& source, Contain& dest,  
    UnaryFunc f) {  
    transform(source.begin(), source.end(), dest.begin(), f);  
}  
  
template<typename Contain1, typename Contain2,  
    typename BinaryFunc>  
void testBinary(Contain1& src1, Contain1& src2,  
    Contain2& dest, BinaryFunc f) {  
    transform(src1.begin(), src1.end(),  
        src2.begin(), dest.begin(), f);  
}  
  
// Executes the expression, then stringizes the  
// expression into the print statement:  
#define T(EXPR) EXPR; print(r.begin(), r.end(), \  
    "After " #EXPR);  
// For Boolean tests:  
#define B(EXPR) EXPR; print(br.begin(), br.end(), \  
    "After " #EXPR);  
  
// Boolean random generator:  
struct BRand {  
    bool operator()() { return rand() % 2 == 0; }  
};  
  
int main() {  
    const int SZ = 10;  
    const int MAX = 50;  
    vector<int> x(SZ), y(SZ), r(SZ);  
    // An integer random number generator:  
    UrandGen urg(MAX);  
    srand(time(0)); // Randomize  
    generate_n(x.begin(), SZ, urg);  
    generate_n(y.begin(), SZ, urg);  
    // Add one to each to guarantee nonzero divide:  
    transform(y.begin(), y.end(), y.begin(),
```



```

    bind2nd(plus<int>(), 1));
// Guarantee one pair of elements is ==:
x[0] = y[0];
print(x.begin(), x.end(), "x");
print(y.begin(), y.end(), "y");
// Operate on each element pair of x & y,
// putting the result into r:
T(testBinary(x, y, r, plus<int>()));
T(testBinary(x, y, r, minus<int>()));
T(testBinary(x, y, r, multiplies<int>()));
T(testBinary(x, y, r, divides<int>()));
T(testBinary(x, y, r, modulus<int>()));
T(testUnary(x, r, negate<int>()));
vector<bool> br(SZ); // For Boolean results
B(testBinary(x, y, br, equal_to<int>()));
B(testBinary(x, y, br, not_equal_to<int>()));
B(testBinary(x, y, br, greater<int>()));
B(testBinary(x, y, br, less<int>()));
B(testBinary(x, y, br, greater_equal<int>()));
B(testBinary(x, y, br, less_equal<int>()));
B(testBinary(x, y, br, not2(greater_equal<int>())));
B(testBinary(x, y, br, not2(less_equal<int>())));
vector<bool> b1(SZ), b2(SZ);
generate_n(b1.begin(), SZ, BRand());
generate_n(b2.begin(), SZ, BRand());
print(b1.begin(), b1.end(), "b1");
print(b2.begin(), b2.end(), "b2");
B(testBinary(b1, b2, br, logical_and<int>()));
B(testBinary(b1, b2, br, logical_or<int>()));
B(testUnary(b1, br, logical_not<int>()));
B(testUnary(b1, br, not1(logical_not<int>())));
} ///:~

```

这个例子使用了一个简单的函数模板 **print()**，它能够打印任意类型的序列并且可以附加可选择的信息。这个模板包含在头文件 **PrintSequence.h** 中，详细内容将在本章后面介绍。

这两个模板函数用来自动处理测试各种函数对象模板的过程。有两个模板函数，是因为函数对象可能是一元的也可能是二元的。**testUnary()** 函数有一个源 **vector**、一个目的 **vector** 和一个用在源 **vector** 上来产生目的 **vector** 的一元函数对象。在 **testBinary()** 中，将两个源 **vector** 传送给一个二元函数来产生目的 **vector**。在这两种情况下，模板函数仅回转并调用 **transform()** 算法，**transform()** 算法将位于其第4个参数位置的一元函数或函数对象应用于序列中的每一元素上，并将结果输出到第3个参数所指示的序列中，在本例中与输入序列相同。

对于每个测试，用户都想看到描述测试的字符串和附加测试结果的字符串。为了自动完成这些工作，可以方便地使用预处理器，宏 **T()** 和 **B()** 分别含有用户想要执行的表达式。对表达式求值后，它们将相应范围的序列传递给 **print()**。为了产生这一信息，通过预处理器将表达式“字符串化” (stringized)。用这种方法，用户就可以看到相应的表达式代码，这些代码在程序执行后存储在结果 **vector** 中。

最后一个小工具 **BRand** 是一个创建随机 **bool** 型值的发生器对象。为了完成这一工作，它从 **rand()** 得到一个随机数并且检查它是否大于  $(\text{RAND\_MAX}+1)/2$ 。如果随机数均匀地分布，则值大于  $(\text{RAND\_MAX}+1)/2$  的情况将以50%的概率出现。

在 **main()** 中，创建了3个 **int** 型的 **vector**：**x** 和 **y** 是源数值，**r** 是结果值。为了用不大于50的随机数初始化 **x** 和 **y**，使用 **Generators.h** 中的 **URandGen** 类型的一个发生器来完成这个任务。标准 **generate\_n()** 算法通过调用第3个参数（必须是一个发生器）、一个给定的次数

(在第2个参数中指定)来建立由第1个参数指定的一个序列。因为有一个 $x$ 被 $y$ 除的操作,所以必须保证 $y$ 的值不为0,以防计算结果溢出。这是靠再次使用`transform()`算法完成的,它从 $y$ 中获得源值并把结果写回 $y$ 。用下面的表达式创建了一个函数对象:

```
bind2nd(plus<int>(), 1)
```

表达式用`plus`函数对象来使第1个参数增1。如同本章前面所做的一样,在这里使用绑定程序适配器来构造一个一元函数,这样仅调用`transform()`就能将其应用到一个序列上。

程序中的另外一个测试是比较两个`vector`中的元素是否相等,因此值得注意的是要保证至少有一对元素是相等的;这里包含0元素。

一旦打印了这两个`vector`,`T()`测试产生数字型值的每一个函数对象,`B()`测试产生布尔型结果的每一个函数对象。在打印`vector`时,将结果放入`vector<bool>`,它对于真值产生'1',对假值产生'0'。下面是执行`FunctionObjects.cpp`的输出结果:

```
x:
4 8 18 36 22 6 29 19 25 47
y:
4 14 23 9 11 32 13 15 44 30
After testBinary(x, y, r, plus<int>()):
8 22 41 45 33 38 42 34 69 77
After testBinary(x, y, r, minus<int>()):
0 -6 -5 27 11 -26 16 4 -19 17
After testBinary(x, y, r, multiplies<int>()):
16 112 414 324 242 192 377 285 1100 1410
After testBinary(x, y, r, divides<int>()):
1 0 0 4 2 0 2 1 0 1
After testBinary(x, y, r, limit<int>()):
0 8 18 0 0 6 3 4 25 17
After testUnary(x, r, negate<int>()):
-4 -8 -18 -36 -22 -6 -29 -19 -25 -47
After testBinary(x, y, br, equal_to<int>()):
1 0 0 0 0 0 0 0 0 0
After testBinary(x, y, br, not_equal_to<int>()):
0 1 1 1 1 1 1 1 1 1
After testBinary(x, y, br, greater<int>()):
0 0 0 1 1 0 1 1 0 1
After testBinary(x, y, br, less<int>()):
0 1 1 0 0 1 0 0 1 0
After testBinary(x, y, br, greater_equal<int>()):
1 0 0 1 1 0 1 1 0 1
After testBinary(x, y, br, less_equal<int>()):
1 1 1 0 0 1 0 0 1 0
After testBinary(x, y, br, not2(greater_equal<int>())):
0 1 1 0 0 1 0 0 1 0
After testBinary(x,y,br,not2(less_equal<int>())):
0 0 0 1 1 0 1 1 0 1
b1:
0 1 1 0 0 0 1 0 1 1
b2:
0 1 1 0 0 0 1 0 1 1
After testBinary(b1, b2, br, logical_and<int>()):
0 1 1 0 0 0 1 0 1 1
After testBinary(b1, b2, br, logical_or<int>()):
0 1 1 0 0 0 1 0 1 1
After testUnary(b1, br, logical_not<int>()):
1 0 0 1 1 1 0 1 0 0
After testUnary(b1, br, not1(logical_not<int>())):
0 1 1 0 0 0 1 0 1 1
```



如果在输出结果中想使布尔型值显示为“真”和“假”而不是1和0，则要调用 **cout.setf(ios::boolalpha)**。

一个绑定程序无需产生一元判定函数；它能创建任意的一元函数（即返回除**bool**型以外类型值的函数）。例如，可以用10乘以**vector**中的每个元素来使用带有绑定程序的 **transform()**算法：

```
//: C06:FBinder.cpp
// Binders aren't limited to producing predicates.
//{L} Generators
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <functional>
#include <iostream>
#include <iterator>
#include <vector>
#include "Generators.h"
using namespace std;

int main() {
    ostream_iterator<int> out(cout, " ");
    vector<int> v(15);
    srand(time(0)); // Randomize
    generate(v.begin(), v.end(), URandGen(20));
    copy(v.begin(), v.end(), out);
    transform(v.begin(), v.end(), v.begin(),
              bind2nd(multiplies<int>(), 10));
    copy(v.begin(), v.end(), out);
} ///:~
```

因为**transform()**的第3个参数与第1个参数一样，所以结果元素又被复制回源**vector**。本例中**bind2nd()**创建的函数对象产生一个**int**型结果。

由绑定程序“绑定”的参数不能是一个函数对象，但也无需是一个编译时常量。例如：

```
//: C06:BinderValue.cpp
// The bound argument can vary.
#include <algorithm>
#include <functional>
#include <iostream>
#include <iterator>
#include <cstdlib>
using namespace std;

int boundedRand() { return rand() % 100; }

int main() {
    const int SZ = 20;
    int a[SZ], b[SZ] = {0};
    generate(a, a + SZ, boundedRand);
    int val = boundedRand();
    int* end = remove_copy_if(a, a + SZ, b,
                             bind2nd(greater<int>(), val));
    // Sort for easier viewing:
    sort(a, a + SZ);
    sort(b, end);
    ostream_iterator<int> out(cout, " ");
    cout << "Original Sequence:" << endl;
    copy(a, a + SZ, out); cout << endl;
    cout << "Values <= " << val << endl;
    copy(b, end, out); cout << endl;
} ///:~
```



这里用20个在0到100之间的随机数填充一个数组，当然用户可以在命令行提供一个值，用来限制产生随机数的范围。在**remove\_copy\_if()**调用中，可以看到对于**bind2nd()**的绑定参数是在相同范围内的顺序序列的随机数。这是一次运行的输出结果：

```
Original Sequence:
4 12 15 17 19 21 26 30 47 48 56 58 60 63 71 79 82 90 92 95
Values <= 41
4 12 15 17 19 21 26 30
```

### 6.2.5 函数指针适配器

算法无论在什么地方都要求有一个类似函数的实体，系统可以提供指向普通函数或是一个函数对象的指针。当算法通过函数指针调用时，就启用了本地的函数调用机制。如果是通过函数对象调用，则执行对象的**operator()**成员。在**CopyInts2.cpp**中，把原始的函数**gt15()**作为一个判定函数传递给**remove\_copy\_if()**。同时也把指向返回随机数的函数的指针传递给**generate()**和**generate\_n()**。

不能通过诸如**bind2nd()**函数对象适配器来使用原始函数，因为这些函数对象适配器要求具有参数及结果类型的类型定义。不需要采用手工方式将原始的函数转化为函数对象，标准库为用户提供了一系列适配器来完成这一工作。**ptr\_fun()**适配器把指向一个函数的指针转化成为一个函数对象。所有这些并不是为无参数函数设计的——也就是说，它们必须是一元或二元函数。

下面的程序用**ptr\_fun()**来封装一个一元函数。

```
//: C06:PtrFun1.cpp
// Using ptr_fun() with a unary function.
#include <algorithm>
#include <cmath>
#include <functional>
#include <iostream>
#include <iterator>
#include <vector>
using namespace std;

int d[] = { 123, 94, 10, 314, 315 };
const int DSZ = sizeof d / sizeof *d;
bool isEven(int x) { return x % 2 == 0; }

int main() {
    vector<bool> vb;
    transform(d, d + DSZ, back_inserter(vb),
        not1(ptr_fun(isEven)));
    copy(vb.begin(), vb.end(),
        ostream_iterator<bool>(cout, " "));
    cout << endl;
    // Output: 1 0 0 0 1
} ///:~
```

不能仅把**isEven**传递给**not1**，因为**not1**需要知道它使用的实际参数的类型和返回值的类型。**ptr\_fun()**适配器可以通过模板参数推断出这些类型。**ptr\_fun()**的一元版本的定义如下所示：

```
template<class Arg, class Result>
pointer_to_unary_function<Arg, Result>
ptr_fun(Result (*fptr)(Arg)) {
    return pointer_to_unary_function<Arg, Result>(fptr);
}
```

正如读者看到的，**ptr\_fun()**的这种版本从**fptr**中推断出参数和结果的类型，并用它们来初始化一个存储**fptr**的**pointer\_to\_unary\_function**对象。如代码的最后一行，对**pointer\_to\_unary\_function**的函数调用操作符仅调用**fptr**：

```
template<class Arg, class Result>
class pointer_to_unary_function
: public unary_function<Arg, Result> {
    Result (*fptr)(Arg); // Stores the f-ptr
public:
    pointer_to_unary_function(Result (*x)(Arg)) : fptr(x) {}
    Result operator()(Arg x) const { return fptr(x); }
};
```

因为**pointer\_to\_unary\_function**派生于**unary\_function**，产生合适的类型定义对**not1**是很有用的。

同时也有**ptr\_fun()**的二元版本，它返回一个在执行上与一元情况类似的**pointer\_to\_binary\_function**对象（派生于**binary\_function**）。下面的程序使用**ptr\_fun()**的二元版本来增加序列中的乘方个数。同时，在向**ptr\_fun()**传递重载函数时也暴露出一个缺陷。

```
//: C06:PtrFun2.cpp {-edg}
// Using ptr_fun() for a binary function.
#include <algorithm>
#include <cmath>
#include <functional>
#include <iostream>
#include <iterator>
#include <vector>
using namespace std;

double d[] = { 01.23, 91.370, 56.661,
              023.230, 19.959, 1.0, 3.14159 };
const int DSZ = sizeof d / sizeof *d;

int main() {
    vector<double> vd;
    transform(d, d + DSZ, back_inserter(vd),
              bind2nd(ptr_fun<double, double, double>(pow), 2.0));
    copy(vd.begin(), vd.end(),
          ostream_iterator<double>(cout, " "));
    cout << endl;
} ///:~
```

**pow()**函数在标准C++头文件**<cmath>**中对每个浮点数据类型进行重载，如下面程序所示：

```
float pow(float, int); // Efficient int power versions ...
double pow(double, int);
long double pow(long double, int);
float pow(float, float);
double pow(double, double);
long double pow(long double, long double);
```

因为有多种**pow()**的版本，编译器不知道选择哪一个。在这里，需要借助前面章节介绍的显式的函数模板特化来帮助编译器。<sup>⊖</sup>

⊖ 对于不同的库实现情况有所不同。如果**pow()**使用C链接，这就意味着读者没有将其理解为C++函数，那么这个例子将不能通过编译。**ptr\_fun**要求是一指向普通重载的C++函数指针。



用通用算法将一个成员函数转化为适于使用的函数对象更是巧妙。例如，假定这里有一个经典的“图形 (shape)”问题，并且想对**Shape**容器内的每个指针都应用**draw()**成员函数：

```
//: C06:MemFun1.cpp
// Applying pointers to member functions.
#include <algorithm>
#include <functional>
#include <iostream>
#include <vector>
#include "../purge.h"
using namespace std;

class Shape {
public:
    virtual void draw() = 0;
    virtual ~Shape() {}
};

class Circle : public Shape {
public:
    virtual void draw() { cout << "Circle::Draw()" << endl; }
    ~Circle() { cout << "Circle::~~Circle()" << endl; }
};

class Square : public Shape {
public:
    virtual void draw() { cout << "Square::Draw()" << endl; }
    ~Square() { cout << "Square::~~Square()" << endl; }
};

int main() {
    vector<Shape*> vs;
    vs.push_back(new Circle);
    vs.push_back(new Square);
    for_each(vs.begin(), vs.end(), mem_fun(&Shape::draw));
    purge(vs);
} ///:~
```

**for\_each()**算法将序列中每一个元素依次传递给由第3个参数指示的函数对象。在这里，希望函数对象封装成它自身类的一个成员函数，所以对于成员函数调用来说，函数对象“参数”成了对象指针。为了产生这样的函数对象，**mem\_fun()**模板使用了指向成员的一个指针来作为它的参数。

**mem\_fun()**函数，是通过传递成员函数所操作的对象的指针作为参数，来产生函数对象；而**mem\_fun\_ref()**函数则直接以对象作为参数。**mem\_fun()**和**mem\_fun\_ref()**都有一组重载版本，用来处理接受零个或一个参数的成员函数，并且它们还分别有一组重载函数用来处理**const**和非**const**成员函数。不过，模板和重载机制负责它们的分类调用，你需要记住的只是何时应该使用**mem\_fun()**而何时又应该使用**mem\_fun\_ref()**。

假设有一个对象（不是指针）的容器，现在想调用有一个参数的成员函数。传递的参数应该来自对象的第2个容器。为了完成这个调用，使用**transform()**算法的第2种重载版本：

```
//: C06:MemFun2.cpp
// Calling member functions through an object reference.
#include <algorithm>
#include <functional>
#include <iostream>
#include <iterator>
#include <vector>
```

```

using namespace std;

class Angle {
    int degrees;
public:
    Angle(int deg) : degrees(deg) {}
    int mul(int times) { return degrees *= times; }
};

int main() {
    vector<Angle> va;
    for(int i = 0; i < 50; i += 10)
        va.push_back(Angle(i));
    int x[] = { 1, 2, 3, 4, 5 };
    transform(va.begin(), va.end(), x,
        ostream_iterator<int>(cout, " "),
        mem_fun_ref(&Angle::mul));
    cout << endl;
    // Output: 0 20 60 120 200
} ///:~

```

因为容器持有对象，所以必须通过成员函数指针来使用 `mem_fun_ref()`。这种版本的 `transform()` 使用第1个范围（对象生存的范围）的开始和结束点；第2个范围的开始点，就是持有的那个表示成员函数的参数；目的迭代器，在本例中就是标准输出；且为每个对象调用函数对象。用 `mem_fun_ref()` 和想要得到的成员指针来创建函数对象。注意，`transform()` 和 `for_each()` 模板函数都是不完全的；`transform()` 要求它调用的函数返回一个值，`for_each()` 没有向它调用的成员函数传递所需的两个参数。因此，不能使用 `transform()` 调用返回值为 `void` 的成员函数，也不能调用只有一个参数的 `for_each()` 成员函数。

大多数任意类型的成员函数与 `mem_fun_ref()` 一起工作。如果用户使用的编译器没有增加任何一个默认参数而超过标准库中指定的正规参数<sup>①</sup>，也可以使用标准库成员函数。例如，假设用户希望读一个文件并且查找其中的空白行。编译器可以允许像下面的程序一样使用 `string::empty()` 成员函数：

```

//: C06:FindBlanks.cpp
// Demonstrates mem_fun_ref() with string::empty().
#include <algorithm>
#include <cassert>
#include <cstddef>
#include <fstream>
#include <functional>
#include <string>
#include <vector>
#include "../require.h"
using namespace std;

typedef vector<string>::iterator LSI;

int main(int argc, char* argv[]) {
    char* fname = "FindBlanks.cpp";
    if(argc > 1) fname = argv[1];
    ifstream in(fname);
    assure(in, fname);
    vector<string> vs;

```

① 如果编译器能够使用默认参数（这些参数是合法的）来定义 `string::empty`，那么表达式 `&string::empty` 就能定义一指向成员函数的指针，这个成员函数包含所有参数。因为无法让编译器提供额外参数，在将算法通过 `mem_fun_ref` 应用于 `string::empty` 时将出现“缺少参数”错误。

```

string s;
while(getline(in, s))
    vs.push_back(s);
vector<string> cpy = vs; // For testing
LSI lsi = find_if(vs.begin(), vs.end(),
    mem_fun_ref(&string::empty));
while(lsi != vs.end()) {
    *lsi = "A BLANK LINE";
    lsi = find_if(vs.begin(), vs.end(),
        mem_fun_ref(&string::empty));
}
for(size_t i = 0; i < cpy.size(); i++)
    if(cpy[i].size() == 0)
        assert(vs[i] == "A BLANK LINE");
    else
        assert(vs[i] != "A BLANK LINE");
} ///:~

```

这个例子使用**find\_if()**、**mem\_fun\_ref()**和**string::empty()**一起，在指定范围的序列中查找第1个空白行的位置。打开文件并将其读入到**vector**对象后，重复这个处理，在文件中查找每一个空白行。每次找到一个空白行时，就用字符串“A BLANK LINE”来取代该空白行。所有这些工作，是在不引用迭代器来选择当前字符串的情况下完成的。

## 6.2.6 编写自己的函数对象适配器

考虑如何编写一个把表示浮点数的字符串转化为相应实际数字值的程序。作为对该问题进行编程的开始，这里有个创建字符串的发生器：

```

//: C06:NumStringGen.h
// A random number generator that produces
// strings representing floating-point numbers.
#ifndef NUMSTRINGGEN_H
#define NUMSTRINGGEN_H
#include <cstdlib>
#include <string>

class NumStringGen {
    const int sz; // Number of digits to make
public:
    NumStringGen(int ssz = 5) : sz(ssz) {}
    std::string operator()() {
        std::string digits("0123456789");
        const int ndigits = digits.size();
        std::string r(sz, ' ');
        // Don't want a zero as the first digit
        r[0] = digits[std::rand() % (ndigits - 1)] + 1;
        // Now assign the rest
        for(int i = 1; i < sz; ++i)
            if(sz >= 3 && i == sz/2)
                r[i] = '.'; // Insert a decimal point
            else
                r[i] = digits[std::rand() % ndigits];
        return r;
    }
};
#endif // NUMSTRINGGEN_H ///:~

```

当创建**NumStringGen**对象时，要告诉它字符串应该有多大。字符串由随机数发生器挑选出来的数字组成，并在中间插入一个小数点。

下面的程序使用**NumStringGen**来填写一个**vector<string>**。但是，要使用标准C库

函数**atof()**来把字符串转化为浮点型数，**string**类型对象必须首先转化为**char**类型指针，这是因为从**string**到**char\***没有自动类型转换。可以使用拥有**mem\_fun\_ref()**和**string::c\_str()**的**transform()**算法，先把所有的**string**都转化为**char\***，然后再使用**atof**进行转换。

```
//: C06:MemFun3.cpp
// Using mem_fun().
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <functional>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>
#include "NumStringGen.h"
using namespace std;

int main() {
    const int SZ = 9;
    vector<string> vs(SZ);
    // Fill it with random number strings:
    srand(time(0)); // Randomize
    generate(vs.begin(), vs.end(), NumStringGen());
    copy(vs.begin(), vs.end(),
        ostream_iterator<string>(cout, "\\t"));
    cout << endl;
    const char* vcp[SZ];
    transform(vs.begin(), vs.end(), vcp,
        mem_fun_ref(&string::c_str));
    vector<double> vd;
    transform(vcp, vcp + SZ, back_inserter(vd),
        std::atof);
    cout.precision(4);
    cout.setf(ios::showpoint);
    copy(vd.begin(), vd.end(),
        ostream_iterator<double>(cout, "\\t"));
    cout << endl;
} ///:~
```

这个程序做了两个转换：一是将C++字符串转换成C风格的字符串（字符数组），另一个是通过**atof()**将C风格的字符串转换成数值。把这两个运算组合成一个将会更好。毕竟，在数学上能组合函数，那么在C++中为什么不能呢？

简明的方法就是用两个函数作为参数并按合适的顺序应用它们：

```
//: C06:ComposeTry.cpp
// A first attempt at implementing function composition.
#include <cassert>
#include <cstdlib>
#include <functional>
#include <iostream>
#include <string>
using namespace std;

template<typename R, typename E, typename F1, typename F2>
class unary_composer {
    F1 f1;
    F2 f2;
public:
    unary_composer(F1 fone, F2 ftwo) : f1(fone), f2(ftwo) {}
```



```

    R operator()(E x) { return f1(f2(x)); }
};

template<typename R, typename E, typename F1, typename F2>
unary_composer<R, E, F1, F2> compose(F1 f1, F2 f2) {
    return unary_composer<R, E, F1, F2>(f1, f2);
}

int main() {
    double x = compose<double, const string&>(
        atof, mem_fun_ref(&string::c_str))("12.34");
    assert(x == 12.34);
} ///:~

```

本例中的**unary\_composer**对象存储函数指针**atof**和**string::c\_str**，这样一来，在调用该对象的**operator()**时首先要应用后面的函数。**compose()**函数适配器是个便利的设置，因此用户无需明确提供全部四个模板参数——**F1**和**F2**从调用中推断出来。

如果无需提供任何模板参数就更好了。这是靠对合适的函数对象坚持类型定义转换来完成的。换句话说，就是假定这些函数的组合是合适的。这就要求为**atof()**使用**ptr\_fun()**。为了使其具有最大的灵活性，一旦把**unary\_composer**传递到一个函数适配器，也能够使其适用。下面的程序这样做了并且很容易地解决了原来的问题：

```

//: C06:ComposeFinal.cpp {-edg}
// An adaptable composer.
#include <algorithm>
#include <cassert>
#include <cstdlib>
#include <functional>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>
#include "NumStringGen.h"
using namespace std;

template<typename F1, typename F2> class unary_composer
: public unary_function<typename F2::argument_type,
                        typename F1::result_type> {
    F1 f1;
    F2 f2;
public:
    unary_composer(F1 f1, F2 f2) : f1(f1), f2(f2) {}
    typename F1::result_type
    operator()(typename F2::argument_type x) {
        return f1(f2(x));
    }
};

template<typename F1, typename F2>
unary_composer<F1, F2> compose(F1 f1, F2 f2) {
    return unary_composer<F1, F2>(f1, f2);
}

int main() {
    const int SZ = 9;
    vector<string> vs(SZ);
    // Fill it with random number strings:
    generate(vs.begin(), vs.end(), NumStringGen());
    copy(vs.begin(), vs.end(),
        ostream_iterator<string>(cout, "\t"));
}

```



```

cout << endl;
vector<double> vd;
transform(vs.begin(), vs.end(), back_inserter(vd),
    compose(ptr_fun(atof), mem_fun_ref(&string::c_str)));
copy(vd.begin(), vd.end(),
    ostream_iterator<double>(cout, "\\t"));
cout << endl;
} ///:~

```

本例中，必须再次使用**typename**来使编译器知道涉及的成员是一个嵌套类型。

一些实现<sup>①</sup>将函数对象的组合作为一个扩展来支持，C++标准委员会可能在标准C++的下一版本中增加这些功能。

### 6.3 STL算法目录

这一部分为读者查找适当的算法提供快捷参考。而把所有的STL算法的完整探究以及问题更深层的细节如性能等一起作为其他参考材料（见本章结尾及附录A）。本节的目标是使读者能够快速熟悉这些算法，并且假定如果需要更多的细节将查找到更多特别指明的参考资料。

尽管读者经常见到用完整的模板声明语法来描述算法，但我们在这里不这样做，因为已经知道它们是模板，并且很容易知道函数声明中的模板参数是什么。参数的类型名为需要的迭代器类型提供描述。读者会发现，这种形式更容易读懂，如果需要，可以很快地在模板头文件中找到所有的声明。

所有涉及迭代器而令人心烦的原因，都是为了（使算法）适用于符合标准库要求的任意类型的容器。到目前为止，本章仅用数组和**vector**作为序列阐述了通用算法，但是在下一章中，读者将会看到一个范围更广的数据结构，这些数据结构支持只用较少力气进行迭代的工作。由于这个原因，将对算法进行部分地分类，这种分类按它们所需要的迭代类型很容易完成。

迭代器的类名描述必须与该迭代器的类型相符合。这些迭代器没有用接口基类来强化这些迭代运算——仅是期望它们在这里出现而已。如有没有接口基类，接收这样程序的编译器可能会抱怨。下面简要地描述迭代器的各种形式：

**InputIterator**。一个只允许单个向序列读入元素的输入迭代器，前向传递使用**operator++**和**operator\***。也可以通过**operator==**和**operator!=**检测输入迭代器。这是约束的范围。

**OutputIterator**。一个只允许单个向序列写入元素的输出迭代器，前向传递使用**operator++**和**operator\***。但是，这类**OutputIterator**不能用**operator==**和**operator!=**来进行测试，因为假定仅持续不断地向目的文件发送元素，而无需判定是否到达了目的文件的结束标志。也就是说，**OutputIterator**涉及的容器可以持有无限个数的对象，而不需要结尾检查。这一点非常重要，因此**OutputIterator**可以与**ostream**（通过**ostream\_iterator**）一起使用，同时也普遍使用“插入”迭代器（它是**back\_inserter()**返回的迭代器类型）。

在相同范围的序列内，没有方法同时确定多个**InputIterators**或**OutputIterators**点，因此也就没有办法一起使用这样的迭代器。仅用迭代器来支持**istream**和**ostream**，使用**InputIterator**和**OutputIterator**就会产生理想的效果。同时也要注意，使用**InputIterators**或**OutputIterators**的算法对可接受的迭代器类型做最弱的限制，这意味着

① 比如Borland C++第6版和Digital Mars编译器都提供的STLPort，而且STLPort基于SGI STL。

当遇到**InputIterator**或**OutputIterator**作为STL算法模板参数时，可以使用任意“更复杂”的迭代器类型。

**ForwardIterator**。因为仅仅可以从**InputIterator**中读和向**OutputIterator**中写，不能用它们中的任何一个来同时读和修改某个范围的数据元素，对这样的迭代器只能解析一次。使用**ForwardIterator**，这些限制就放松了；仍然仅用**operator++**前向移动，但是可以同时读和写，并且可以在相同的范围内比较这些迭代器是否相等。因为前向迭代器可以同时读和写，可以用它来取代**InputIterator**或**OutputIterator**。

**BidirectionalIterator**。实际上，这是一个也可以进行后向移动的**ForwardIterator**。也就是说，**BidirectionalIterator**支持**ForwardIterator**所做的全部操作，而另外还增加了**operator--**运算。

**RandomAccessIterator**。这种迭代器类型支持一个常规指针所做的全部运算：可以通过增加和减少某个整数值，来向前和向后跳跃移动（不是每次只移动一个元素），还可以用**operator[]**作为下标索引，可以从一个迭代器中减去另一个迭代器，也可以用**operator<**，**operator>**来比较迭代器看哪个更大等等。如果来实现一个排序程序或其他类似的工作，随机存取迭代器是创建一个有效率的算法所必需的。

本章后面的算法描述中，使用的模板参数类型名由列出的迭代器类型（有时附加‘1’或‘2’来区分不同的模板参数）组成，同时也包括其他的参数，通常是函数对象。

当描述传递给运算的元素组时，经常使用数学上“范围”记号。即方括号表示“包括边界点”，圆括号表示“不包括边界点”。当使用迭代器时，要靠指向开始元素的迭代器和指向超越最后一个元素的“超越末尾的”迭代器来决定一个范围。由于根本就没有使用超越末尾的元素，决定这样一对迭代器的范围可以表示成**[first,last)**，这里**first**是指向开始元素的迭代器，**last**是超越末尾的迭代器。

大多数教材和对STL算法的讨论都根据它们副作用的大小来组织算法的先后顺序：非变异（non-mutating）算法在作用域范围内不对元素进行改变，变异（mutating）算法改变元素，等等。这些描述基于主要的基础行为或算法的实现——也就是基于设计者的观点。在实际使用中，用户会发现这种分类没用，因此应该根据要解决的问题来组织算法：当你查找某个元素或元素集合时，是不是对每个元素都执行一个运算、计算元素个数并且更新元素等等？这应该有助于更容易发现要求的算法。

如果在函数的声明前面没看到一个如<utility>或<numeric>的头文件，那么它就应该出现在<algorithm>中。同样地，所有的算法都在名字空间std中。

### 6.3.1 实例创建的支持工具

创建一些基本的工具来测试算法是很有用的。在这些例子中，将使用前面在**Generators.h**中涉及的发生器以及下面出现的这些内容。

显示一个序列是经常要做的工作，这里有一个函数模板用来打印任意一个序列，它不考虑序列中包含的数据类型：

```
//: C06:PrintSequence.h
// Prints the contents of any sequence.
#ifndef PRINTSEQUENCE_H
#define PRINTSEQUENCE_H
#include <algorithm>
#include <iostream>
#include <iterator>
```

```

template<typename Iter>
void print(Iter first, Iter last, const char* nm = "",
          const char* sep = "\n",
          std::ostream& os = std::cout) {
    if(nm != 0 && *nm != '\0')
        os << nm << ": " << sep;
    typedef typename
        std::iterator_traits<Iter>::value_type T;
    std::copy(first, last,
              std::ostream_iterator<T>(std::cout, sep));
    os << std::endl;
}
#endif // PRINTSEQUENCE_H ///:~

```

在默认的情况下，以一个换行符（“**n**”）作为分隔符，这个函数模板向**cout**输出，但可以通过修改默认参数来改变它。同时也可以输出的开头打印一个信息。因为**print()**使用**copy()**算法经由**ostream\_iterator**向**cout**发送对象，**ostream\_iterator**必须知道它正在打印的对象的类型，该对象的类型由传递过来的迭代器的**value\_type**成员推断而来。

**std::iterator\_traits**模板能够使**print()**函数模板处理由任意迭代器类型限定的序列。由标准容器如**vector**返回的迭代器类型定义了一个嵌套类型**value\_type**，它代表元素的类型。但是当使用数组时，迭代器仅仅只是指针类型，因而不能有嵌套类型。为了支持标准库中与迭代器有关联的使用便利的类型，**std::iterator\_traits**为指针类型提供了下面的半特化：

```

template<class T>
struct iterator_traits<T*> {
    typedef random_access_iterator_tag iterator_category;
    typedef T value_type;
    typedef ptrdiff_t difference_type;
    typedef T* pointer;
    typedef T& reference;
};

```

这样就使该模板可获得经由类型名**value\_type**指明的元素类型（即**T**）。

#### 稳定排序和不稳定排序

对于很多经常移动序列中元素的STL算法而言，有序列的稳定再排序和不稳定再排序之分。就比较函数而言，一个稳定的排序保持相等元素的原始相对顺序。例如，考虑序列{**c(1)**, **b(1)**, **c(2)**, **a(1)**, **b(2)**, **a(2)**}。在算法中是根据字母来检查这些元素的相等性，但是它们的数字显示怎样在序列中出现（谁在前，谁在后？）。如果排序（例如），对这个序列使用不稳定的排序，就不能保证相同字母间的特定顺序，所以可能以{**a(2)**, **a(1)**, **b(1)**, **b(2)**, **c(2)**, **c(1)**}结束。然而，如果使用稳定的排序，就会得到{**a(1)**, **a(2)**, **b(1)**, **b(2)**, **c(1)**, **c(2)**}。STL的**sort()**算法使用的是快速排序的一个变种，因此是不稳定的，但是STL也提供稳定的排序算法**stable\_sort()**。<sup>⊖</sup>

为了证明对一个序列进行重新排序的算法是稳定性算法还是不稳定性算法，我们需要一些方法来保持对元素原始位置的跟踪。下面是一种保持跟踪特殊对象原始出现顺序的**string**对象，它用**static map**对从**NString**到**Counters**进行映射。这样每个**NString**包含一个**occurrence**字段，用来表示在**NString**中发现的顺序。

⊖ **stable\_sort()**使用归并排序，归并排序是稳定排序，但是在平均情况下比快速排序慢。



```

//: C06:NString.h
// A "numbered string" that keeps track of the
// number of occurrences of the word it contains.
#ifndef NSTRING_H
#define NSTRING_H
#include <algorithm>
#include <iostream>
#include <string>
#include <utility>
#include <vector>

typedef std::pair<std::string, int> psi;

// Only compare on the first element
bool operator==(const psi& l, const psi& r) {
    return l.first == r.first;
}

class NString {
    std::string s;
    int thisOccurrence;
    // Keep track of the number of occurrences:
    typedef std::vector<psi> vp;
    typedef vp::iterator vpit;
    static vp words;
    void addString(const std::string& x) {
        psi p(x, 0);
        vpit it = std::find(words.begin(), words.end(), p);
        if(it != words.end())
            thisOccurrence = ++it->second;
        else {
            thisOccurrence = 0;
            words.push_back(p);
        }
    }
public:
    NString() : thisOccurrence(0) {}
    NString(const std::string& x) : s(x) { addString(x); }
    NString(const char* x) : s(x) { addString(x); }
    // Implicit operator= and copy-constructor are OK here.
    friend std::ostream& operator<<(
        std::ostream& os, const NString& ns) {
        return os << ns.s << " [" << ns.thisOccurrence << "]\n";
    }
    // Need this for sorting. Notice it only
    // compares strings, not occurrences:
    friend bool
    operator<(const NString& l, const NString& r) {
        return l.s < r.s;
    }
    friend
    bool operator==(const NString& l, const NString& r) {
        return l.s == r.s;
    }
    // For sorting with greater<NString>:
    friend bool
    operator>(const NString& l, const NString& r) {
        return l.s > r.s;
    }
    // To get at the string directly:
    operator const std::string&() const { return s; }
};

```



```
// Because NString::vp is a template and we are using the
// inclusion model, it must be defined in this header file:
NString::vp NString::words;
#endif // NSTRING_H ///:~
```

通常使用**map**容器来将一个与字符串一起出现的数字关联起来，但直到第7章才会讨论映射的问题，因此在这里用成对的**vector**来代替映射。在第7章读者将会看到大量类似的例子。

执行有秩序的升序排序必须的运算符只有**NString::operator<()**。同时还提供降序的排序操作符**operator>()**，这样**greater**模板就能调用它了。

### 6.3.2 填充和生成

这些算法能够自动用一个特定值来填充（容器中的）某个范围的数据，或为（容器中的）某个特定范围生成一组值。“填充（fill）”函数向容器中多次插入一个值。“生成（generate）”函数使用如前面提到过的发生器来产生插入到容器中的值。

```
void fill(ForwardIterator first, ForwardIterator last,
          const T& value);
void fill_n(OutputIterator first, Size n, const T& value);
```

**fill()**对 **[first,last)** 范围内的每个元素赋值**value**。**fill\_n()**对由**first**开始的**n**个元素赋值**value**。

```
void generate(ForwardIterator first, ForwardIterator last,
              Generator gen);
void generate_n(OutputIterator first, Size n, Generator
                gen);
```

**generate()**为 **[first,last)** 范围内的每个元素进行一个**gen()**调用，可以假定为每个元素产生一个不同的值。**generate\_n()**对**gen()**调用**n**次，并且将返回值赋给由**first**开始的**n**个元素。

程序举例

下面的例子对**vector**进行填充和生成。同时也显示了**print()**的使用：

```
//: C06:FillGenerateTest.cpp
// Demonstrates "fill" and "generate."
//{L} Generators
#include <vector>
#include <algorithm>
#include <string>
#include "Generators.h"
#include "PrintSequence.h"
using namespace std;

int main() {
    vector<string> v1(5);
    fill(v1.begin(), v1.end(), "howdy");
    print(v1.begin(), v1.end(), "v1", " ");
    vector<string> v2;
    fill_n(back_inserter(v2), 7, "bye");
    print(v2.begin(), v2.end(), "v2");
    vector<int> v3(10);
    generate(v3.begin(), v3.end(), SkipGen(4,5));
    print(v3.begin(), v3.end(), "v3", " ");
    vector<int> v4;
    generate_n(back_inserter(v4), 15, URandGen(30));
    print(v4.begin(), v4.end(), "v4", " ");
} ///:~
```



**vector<string>** 用预定义的大小来创建。因为已经为**vector**中所有**string**对象创建了存储空间，**fill()**可以用它的赋值操作对**vector**中的每个空间赋“howdy”的一个拷贝。同时，用空格来取代默认的换行符分隔符。

没有给定第2个**vector<string> v2**的初始大小，因此必须使用**back\_inserter()**来添加新元素，而不是试图对现有位置赋值。

除了用一个发生器来代替常量值以外，**generate()**和**generate\_n()**函数与“填充”函数有相同的形式。在这里，这两个发生器都演示了它们的功能。

### 6.3.3 计数

所有的容器都含有一个成员函数**size()**，它可以告之该容器包含有多少个元素。**size()**的返回类型是迭代器的**difference\_type**<sup>①</sup>（通常是**ptrdiff\_t**），下面我们用**Integral Value**表示。下面的两个算法可以满足一定标准的对象计数。

```
IntegralValue count(InputIterator first, InputIterator
    last, const EqualityComparable& value);
```

在这个算法中，产生 **[first,last)** 范围内其值等于**value**（当用**operator==**测试时）的元素的个数。

```
IntegralValue count_if(InputIterator first, InputIterator
    last, Predicate pred);
```

这个算法产生 **[first,last)** 范围内能使**pred**返回**true**的元素的个数。

程序举例

这里，用随机字符（包括一些重复的字符）填充**vector<char> v**。**set<char>**由**v**来初始化，因此它仅持有**v**中代表的各个字母中的一个。这个**set**对显示的所有字符的实例进行计数：

```
//: C06:Counting.cpp
// The counting algorithms.
//{L} Generators
#include <algorithm>
#include <functional>
#include <iterator>
#include <set>
#include <vector>
#include "Generators.h"
#include "PrintSequence.h"
using namespace std;

int main() {
    vector<char> v;
    generate_n(back_inserter(v), 50, CharGen());
    print(v.begin(), v.end(), "v", "");
    // Create a set of the characters in v:
    set<char> cs(v.begin(), v.end());
    typedef set<char>::iterator sci;
    for(sci it = cs.begin(); it != cs.end(); it++) {
        int n = count(v.begin(), v.end(), *it);
        cout << *it << ": " << n << ", ";
    }
    int lc = count_if(v.begin(), v.end(),
        bind2nd(greater<char>(), 'a'));
    cout << "\nLowercase letters: " << lc << endl;
```

① 在第7章中我们将详细讨论迭代器。



```

    sort(v.begin(), v.end());
    print(v.begin(), v.end(), "sorted", "");
} ///:~

```

**count\_if()** 算法通过对所有的小写字母计数来进行演示；用 **bind2nd()** 和 **greater** 函数对象模板创建判定函数。

#### 6.3.4 操作序列

这些都是有关移动序列的算法。

```

OutputIterator copy(InputIterator first, InputIterator
    last, OutputIterator destination);

```

使用赋值，从范围 **[first,last)** 复制序列到 **destination**，每次赋值后都增加 **destination**。这本质上是一个“左混洗 (shuffle-left)”运算，所以源序列不能包含目的序列。由于使用了赋值操作，因此不能直接向空容器或容器末尾插入元素，而必须把 **destination** 迭代器封装在 **insert\_iterator** 里（在与容器发生联系的情况下，典型地使用 **back\_inserter()** 或 **inserter()**）。

```

BidirectionalIterator2 copy_backward(BidirectionalIterator1
    first, BidirectionalIterator1 last,
    BidirectionalIterator2 destinationEnd);

```

这个算法如同 **copy()** 一样，但是以相反的顺序复制元素。这本质上是“右混洗 (shuffle-right)”运算，而且如同 **copy()** 一样，源序列不能包含目的序列。将源范围 **[first, last)** 序列复制到目的序列，但第1个目的元素是 **destinationEnd - 1**。这个迭代器在每次赋值后减少。目的序列范围的空间必须已经存在（允许赋值），而且目的序列范围不能在源序列范围之内。

```

void reverse(BidirectionalIterator first,
    BidirectionalIterator last);
OutputIterator reverse_copy(BidirectionalIterator first,
    BidirectionalIterator last, OutputIterator destination);

```

这个函数的两种形式都倒置了范围 **[first,last)**。**reverse()** 倒置原序列范围的元素，**reverse\_copy()** 保持原序列范围元素顺序不变，而将倒置的元素复制到 **destination**，返回结果序列范围的超越末尾 (past-the-end) 的迭代器。

```

ForwardIterator2 swap_ranges(ForwardIterator1 first1,
    ForwardIterator1 last1, ForwardIterator2 first2);

```

通过交换对应的元素来交换相等大小两个范围的内容。

```

void rotate(ForwardIterator first, ForwardIterator middle,
    ForwardIterator last);
OutputIterator rotate_copy(ForwardIterator first,
    ForwardIterator middle, ForwardIterator last,
    OutputIterator destination);

```

该算法把 **[first, middle)** 范围中的内容移到该序列的末尾，并且将 **[middle,last)** 范围中的内容移到该序列的开始位置。使用 **rotate()** 在适当的位置执行交换；使用 **rotate\_copy()** 不改变原始序列范围，且将轮换后 (rotated) 版本的元素复制到 **destination**，返回结果范围的超越末尾的迭代器。注意，需要使用 **swap\_ranges()** 时，两个范围的大小是完全相等的，但“轮换”函数不是这样。

```

bool next_permutation(BidirectionalIterator first,
    BidirectionalIterator last);
bool next_permutation(BidirectionalIterator first,
    BidirectionalIterator last, StrictWeakOrdering
    binary_pred);
bool prev_permutation(BidirectionalIterator first,
    BidirectionalIterator last);
bool prev_permutation(BidirectionalIterator first,
    BidirectionalIterator last, StrictWeakOrdering
    binary_pred);

```

在这些算法中，排列(permutation)是一组元素的一种独一无二的排序。如果有 $n$ 个元素，就会有 $n!$ （ $n$ 的阶乘）种不同的元素的组合。所有的这些组合都可以概念化地以词典编纂（像字典一样）的顺序对序列进行排序，这样就产生了一种“后继(next)”和“前驱(previous)”排列的概念。因此无论范围内当前元素的顺序是什么样，在排列的序列中都有一个不同的“后继”和“前驱”的排列。

**next\_permutation()**和**prev\_permutation()**函数对元素重新排列成后继的或前驱的排列，如果成功则返回**true**。如果没有多个“后继”排列，元素以升序排序，**next\_permutation()**返回**false**。如果没有多个“前驱”排列，元素以降序排序，**previous\_permutation()**返回**false**。

具有**StrictWeakOrdering**参数的函数形式用**binary\_pred**来执行比较，而不是**operator<**。

```

void random_shuffle(RandomAccessIterator first,
    RandomAccessIterator last);
void random_shuffle(RandomAccessIterator first,
    RandomAccessIterator last RandomNumberGenerator& rand);

```

这个函数随机地重排范围内的元素。如果用随机数发生器，它会产生均匀的分布结果。第1种形式使用内部随机数发生器，第2种使用用户提供的随机数发生器。对于正数 $n$ 发生器必须返回一个在 $[0, n)$ 范围内的值。

```

BidirectionalIterator partition(BidirectionalIterator
    first, BidirectionalIterator last, Predicate pred);
BidirectionalIterator
stable_partition(BidirectionalIterator first,
    BidirectionalIterator last, Predicate pred);

```

在这些算法中，“划分”函数将满足**pred**的元素移到序列的开始位置。迭代器指向其返回元素位置，该元素是超越这些元素中的最后一个（对于以满足**pred**的元素为开始的子序列，“末尾”迭代器有效）。这个位置通常称为“划分点(partition point)”。

使用**partition()**，在函数调用后，每个结果子序列的元素顺序并没有被指定，但是用**stable\_partition()**，划分点前后这些元素的相对顺序与划分处理前相同。

程序举例

这里给出了序列运算的演示：

```

//: C06:Manipulations.cpp
// Shows basic manipulations.
//{L} Generators
// NString
#include <vector>

```

```

#include <string>
#include <algorithm>
#include "PrintSequence.h"
#include "NString.h"
#include "Generators.h"
using namespace std;

int main() {
    vector<int> v1(10);
    // Simple counting:
    generate(v1.begin(), v1.end(), SkipGen());
    print(v1.begin(), v1.end(), "v1", " ");
    vector<int> v2(v1.size());
    copy_backward(v1.begin(), v1.end(), v2.end());
    print(v2.begin(), v2.end(), "copy_backward", " ");
    reverse_copy(v1.begin(), v1.end(), v2.begin());
    print(v2.begin(), v2.end(), "reverse_copy", " ");
    reverse(v1.begin(), v1.end());
    print(v1.begin(), v1.end(), "reverse", " ");
    int half = v1.size() / 2;
    // Ranges must be exactly the same size:
    swap_ranges(v1.begin(), v1.begin() + half,
        v1.begin() + half);
    print(v1.begin(), v1.end(), "swap_ranges", " ");
    // Start with a fresh sequence:
    generate(v1.begin(), v1.end(), SkipGen());
    print(v1.begin(), v1.end(), "v1", " ");
    int third = v1.size() / 3;
    for(int i = 0; i < 10; i++) {
        rotate(v1.begin(), v1.begin() + third, v1.end());
        print(v1.begin(), v1.end(), "rotate", " ");
    }
    cout << "Second rotate example:" << endl;
    char c[] = "aabbccddeeffgghhiijj";
    const char CSZ = strlen(c);
    for(int i = 0; i < 10; i++) {
        rotate(c, c + 2, c + CSZ);
        print(c, c + CSZ, "", "");
    }
    cout << "All n! permutations of abcd:" << endl;
    int nf = 4 * 3 * 2 * 1;
    char p[] = "abcd";
    for(int i = 0; i < nf; i++) {
        next_permutation(p, p + 4);
        print(p, p + 4, "", "");
    }
    cout << "Using prev_permutation:" << endl;
    for(int i = 0; i < nf; i++) {
        prev_permutation(p, p + 4);
        print(p, p + 4, "", "");
    }
    cout << "random_shuffling a word:" << endl;
    string s("hello");
    cout << s << endl;
    for(int i = 0; i < 5; i++) {
        random_shuffle(s.begin(), s.end());
        cout << s << endl;
    }
    NString sa[] = { "a", "b", "c", "d", "a", "b",
        "c", "d", "a", "b", "c", "d", "a", "b", "c" };
    const int SASZ = sizeof sa / sizeof *sa;
    vector<NString> ns(sa, sa + SASZ);
    print(ns.begin(), ns.end(), "ns", " ");
}

```



```

vector<NString>::iterator it =
    partition(ns.begin(), ns.end(),
        bind2nd(greater<NString>(), "b"));
cout << "Partition point: " << *it << endl;
print(ns.begin(), ns.end(), "", " ");
// Reload vector:
copy(sa, sa + SASZ, ns.begin());
it = stable_partition(ns.begin(), ns.end(),
    bind2nd(greater<NString>(), "b"));
cout << "Stable partition" << endl;
cout << "Partition point: " << *it << endl;
print(ns.begin(), ns.end(), "", " ");
} ///:~

```

观察这个程序结果的最好方法是运行该程序。(也可以将结果重新输出到一个文件中。)

**vector<int> v1**初始化为一个简单的升序序列，并且打印这个序列。读者将会看到**copy\_backward()**的效果（复制到**v2**，**v2**与**v1**相同大小）与普通的复制相同。再一次强调，**copy\_backward()**与**copy()**做相同的工作——只是以相反的顺序操作。

**reverse\_copy()**实际上创建一个相反顺序的复制，**reverse()**在适当的位置执行颠倒操作。接下来，**swap\_ranges()**将颠倒序列的上半部分和下半部分进行交换。范围可以比整个**vector**的子集小，只要它们大小相等就可以。

**rotate()**是重新创建一个升序序列的演示，它通过多次交换**v1**的三分之一来完成排序工作。第2个**rotate()**例子使用了字符且每次仅交换两个字符。通过这个例子，也展示了STL算法和**print()**模板的灵活性，因为与使用其他任意类型一样，可以很容易地使用**char**数组。

为了演示**next\_permutation()**和**prev\_permutation()**，用全部**n!**（**n**的阶乘）种组合来排列“abcd”四个字母的集合。从输出结果中可以看到，排列遵循严格的定义顺序（即排列是确定性的处理）。

**random\_shuffle()**的一个快速演示是将其应用到一个**string**，并且看结果是什么。因为**string**对象含有可以返回合适迭代器的**begin()**和**end()**成员函数，很多STL算法都可以很容易地使用它。同时这里也使用了**char**型数组。

最后，用**NString**数组演示了**partition()**和**stable\_partition()**。读者将会注意到，总计的初始化表达式使用的是**char**型数组，但**NString**含有一个**char\***的构造函数，它能自动调用。

从输出结果中可以看到使用不稳定的划分，对象能正确地“被划分”在划分点之上和之下，但不是以特定的顺序进行；反之用稳定的划分则保持原始的顺序。

### 6.3.5 查找和替换

所有这些算法都用来在某个范围内查找一个或多个对象，该范围由开始的两个迭代器参数定义。

```

InputIterator find(InputIterator first, InputIterator last,
    const EqualityComparable& value);

```

这个算法在某个范围内的序列元素中查找**value**。返回一个迭代器，该迭代器指向在范围**[first, last)**内**value**第1次出现的位置。如果**value**不在范围内，**find()**返回**last**。这是线性查找（linear search）；也就是说，从范围的起始点开始，对每个连续的元素依次进行检查，而不对元素的顺序路径做任何假设。相反，**binary\_search()**（在后面定义）是在一个已经有序的序列上工作，因此能够更快地进行查找。

```
InputIterator find_if(InputIterator first, InputIterator
    last, Predicate pred);
```

这个算法如同**find()**一样，**find\_if()**在指定的序列范围内执行线性查找。然而，代替查找**value**，**find\_if()**寻找一个满足**pred**的元素，当查找到这样的元素时**Predicate pred**返回**true**。如果不能找到这样的元素则返回**last**。

```
ForwardIterator adjacent_find(ForwardIterator first,
    ForwardIterator last);
ForwardIterator adjacent_find(ForwardIterator first,
    ForwardIterator last, BinaryPredicate binary_pred);
```

如同**find()**一样，这些算法在指定的序列范围内执行线性查找。但不是仅查找一个元素，而是查找两个邻近的相等元素。函数的第1种形式查找两个相等的元素（通过**operator==**）。第2种形式查找两个邻近的元素，当找到这两个元素并一起传递给**binary\_pred**时，产生**true**结果。如果找到这样的一对元素，则返回指向两个元素中第1个元素的迭代器，否则返回**last**。

```
ForwardIterator1 find_first_of(ForwardIterator1 first1,
    ForwardIterator1 last1, ForwardIterator2 first2,
    ForwardIterator2 last2);
ForwardIterator1 find_first_of(ForwardIterator1 first1,
    ForwardIterator1 last1, ForwardIterator2 first2,
    ForwardIterator2 last2, BinaryPredicate binary_pred);
```

如同**find()**一样，上面这两个算法也在指定的序列范围内执行线性查找。这两种形式都是在第2个范围内查找与第1个范围内的某个元素相等的元素。第1种形式使用**operator==**，第2种形式使用提供的判定函数。在第2种形式中，第1个范围序列的当前元素成为**binary\_pred**的第1个参数，第2个范围序列内的元素成为**binary\_pred**的第2个参数。

```
ForwardIterator1 search(ForwardIterator1 first1,
    ForwardIterator1 last1, ForwardIterator2 first2,
    ForwardIterator2 last2);
ForwardIterator1 search(ForwardIterator1 first1,
    ForwardIterator1 last1, ForwardIterator2 first2,
    ForwardIterator2 last2, BinaryPredicate binary_pred);
```

这些算法检查第2个序列范围是否出现在第1个序列的范围内（顺序也完全一致），如果是则返回一个迭代器，该迭代器指向在第1个范围序列中第2个范围序列出现的开始位置。如果没有找到就返回**last1**。第1种形式测试使用**operator==**，第2种形式检测被比较的每对元素是否能使**binary\_pred**返回**true**。

```
ForwardIterator1 find_end(ForwardIterator1 first1,
    ForwardIterator1 last1, ForwardIterator2 first2,
    ForwardIterator2 last2);
ForwardIterator1 find_end(ForwardIterator1 first1,
    ForwardIterator1 last1, ForwardIterator2 first2,
    ForwardIterator2 last2, BinaryPredicate binary_pred);
```

这些算法的形式和参数如同**search()**，查找第2个范围的序列是否在第1个范围内作为子集出现，但是**search()**查找该子集首先出现的位置，而**find\_end()**则查找该子集最后出现的位置，并且返回指向该子集的第一个元素的迭代器。

```
ForwardIterator search_n(ForwardIterator first,
    ForwardIterator last, Size count, const T& value);
ForwardIterator search_n(ForwardIterator first,
```



```
ForwardIterator last, Size count, const T& value,
BinaryPredicate binary_pred);
```

这些算法在**[first,last)** 范围内查找一组共**count**个连续的值，这些值都与**value**相等（在第1种形式中），或是当将所有这些与**value**相同的值传递给**binary\_pred**时返回**true**（在第2种形式中）。如果不能找到这样的一组数值就返回**last**。

```
ForwardIterator min_element(ForwardIterator first,
ForwardIterator last);
ForwardIterator min_element(ForwardIterator first,
ForwardIterator last, BinaryPredicate binary_pred);
```

这些算法返回一个迭代器，该迭代器指向范围内“最小的”值首次出现的位置（如下面的解释——范围内可能会多次出现这个值）。如果范围为空则返回**last**。第1种版本用**operator<**执行比较，且返回值为**r**，其意义是：对于范围 **[first,r)** 中每个元素**e**，**\*e < \*r**都为假。第2种版本用**binary\_pred**比较，且返回值为**r**，其意义是：对于范围 **[first,r)** 中每个元素**e**，**binary\_pred(\*e,\*r)**都为假。

```
ForwardIterator max_element(ForwardIterator first,
ForwardIterator last);
ForwardIterator max_element(ForwardIterator first,
ForwardIterator last, BinaryPredicate binary_pred);
```

这些算法返回一个迭代器，该迭代器指向范围内最大值首次出现的位置。（范围内可能会多次出现最大值。）如果范围为空返回**last**。第1种版本用**operator<**执行比较，且返回值为**r**，其意义是：对于范围 **[first,r)** 中每个元素**e**，**\*r < \*e**都为假。第2种版本用**binary\_pred**执行比较，且返回值为**r**，其意义是：对于范围 **[first,r)** 中每个元素**e**，**binary\_pred(\*r,\*e)**都为假。

```
void replace(ForwardIterator first, ForwardIterator last,
const T& old_value, const T& new_value);
void replace_if(ForwardIterator first, ForwardIterator
last, Predicate pred, const T& new_value);
OutputIterator replace_copy(InputIterator first,
InputIterator last, OutputIterator result, const T&
old_value, const T& new_value);
OutputIterator replace_copy_if(InputIterator first,
InputIterator last, OutputIterator result, Predicate
pred, const T& new_value);
```

在这些算法中，每一种“替换”形式都从头至尾在范围 **[first,last)** 内进行查找，找到与标准匹配的值并用**new\_value**替换它们。**replace( )**和**replace\_copy( )**都是仅仅查找**old\_value**并对其进行替换；**replace\_if( )**和**replace\_copy\_if( )**查找满足判定函数**pred**的值。函数的“复制”形式不修改原始范围，而是将作为替代的一个副本赋给**result**，更换它的值（每次赋值后增加**result**）。

#### 程序举例

为了提供简单的可视结果，这个例子运算**int**型的**vector**。再强调一次，并不是将每一个算法的所有版本都展现出来。（一些意义很明显的算法被略去了。）

```
//: C06:SearchReplace.cpp
// The STL search and replace algorithms.
#include <algorithm>
#include <functional>
```

```

#include <vector>
#include "PrintSequence.h"
using namespace std;

struct PlusOne {
    bool operator()(int i, int j) { return j == i + 1; }
};

class MulMoreThan {
    int value;
public:
    MulMoreThan(int val) : value(val) {}
    bool operator()(int v, int m) { return v * m > value; }
};

int main() {
    int a[] = { 1, 2, 3, 4, 5, 6, 6, 7, 7, 7,
               8, 8, 8, 11, 11, 11, 11, 11 };
    const int ASZ = sizeof a / sizeof *a;
    vector<int> v(a, a + ASZ);
    print(v.begin(), v.end(), "v", " ");
    vector<int>::iterator it = find(v.begin(), v.end(), 4);
    cout << "find: " << *it << endl;
    it = find_if(v.begin(), v.end(),
                 bind2nd(greater<int>(), 8));
    cout << "find_if: " << *it << endl;
    it = adjacent_find(v.begin(), v.end());
    while(it != v.end()) {
        cout << "adjacent_find: " << *it
              << ", " << *(it + 1) << endl;
        it = adjacent_find(it + 1, v.end());
    }
    it = adjacent_find(v.begin(), v.end(), PlusOne());
    while(it != v.end()) {
        cout << "adjacent_find PlusOne: " << *it
              << ", " << *(it + 1) << endl;
        it = adjacent_find(it + 1, v.end(), PlusOne());
    }
    int b[] = { 8, 11 };
    const int BSZ = sizeof b / sizeof *b;
    print(b, b + BSZ, "b", " ");
    it = find_first_of(v.begin(), v.end(), b, b + BSZ);
    print(it, it + BSZ, "find_first_of", " ");
    it = find_first_of(v.begin(), v.end(),
                       b, b + BSZ, PlusOne());
    print(it, it + BSZ, "find_first_of PlusOne", " ");
    it = search(v.begin(), v.end(), b, b + BSZ);
    print(it, it + BSZ, "search", " ");
    int c[] = { 5, 6, 7 };
    const int CSZ = sizeof c / sizeof *c;
    print(c, c + CSZ, "c", " ");
    it = search(v.begin(), v.end(), c, c + CSZ, PlusOne());
    print(it, it + CSZ, "search PlusOne", " ");
    int d[] = { 11, 11, 11 };
    const int DSZ = sizeof d / sizeof *d;
    print(d, d + DSZ, "d", " ");
    it = find_end(v.begin(), v.end(), d, d + DSZ);
    print(it, v.end(), "find_end", " ");
    int e[] = { 9, 9 };
    print(e, e + 2, "e", " ");
    it = find_end(v.begin(), v.end(), e, e + 2, PlusOne());
    print(it, v.end(), "find_end PlusOne", " ");
    it = search_n(v.begin(), v.end(), 3, 7);

```

```

print(it, it + 3, "search_n 3, 7", " ");
it = search_n(v.begin(), v.end(),
    6, 15, MulMoreThan(100));
print(it, it + 6,
    "search_n 6, 15, MulMoreThan(100)", " ");
cout << "min_element: "
    << *min_element(v.begin(), v.end()) << endl;
cout << "max_element: "
    << *max_element(v.begin(), v.end()) << endl;
vector<int> v2;
replace_copy(v.begin(), v.end(),
    back_inserter(v2), 8, 47);
print(v2.begin(), v2.end(), "replace_copy 8 -> 47", " ");
replace_if(v.begin(), v.end(),
    bind2nd(greater_equal<int>(), 7), -1);
print(v.begin(), v.end(), "replace_if >= 7 -> -1", " ");
} ///:~

```

这个例子以两个判定函数开始：**PlusOne**是一个二元判定函数，如果第2个参数等于第1个参数加1则返回**true**；**MulMoreThan**也是一个二元判定函数，如果第1个参数与第2个参数的乘积大于存储在对象中的值则返回**true**。这些二元判定函数在例子中用来进行测试。

在**main()**中，创建一个数组**a**并将其提供给**vector<int> v**的构造函数。**vector**是查找和替换行动的目标，注意这里有很多重复元素——它们由一些查找/替换程序发现。

第1个测试演示**find()**，在**v**中发现值4。返回值是指向4的第1个实例的迭代器，如果没有找到要查找的值，返回值指向输入范围的末尾（**v.end()**）。

**find\_if()**算法使用了一个判定函数来决定是否找到了正确的元素。在本例中，用一个动态的**greater<int>**（即，“查看第1个**int**型参数是否大于第2个参数”）和固定的第2个参数8来创建一个执行中的判定函数**bind2nd()**。因此，如果**v**中的值大于8则返回真。

因为在许多情况下**v**中会出现两个相同的相邻对象，所以设计**adjacent\_find()**测试来找到它们。查找从序列的开始位置出发，然后进入一个**while**循环，以便确定迭代器**it**没有到达该输入序列的末尾（这意味着不能再找到更多匹配的元素）。对于找到的每个匹配，循环打印这些匹配的元素并且执行下一个**adjacent\_find()**，这时就使用**it+1**作为第1个参数（用这种方式在**三元组**中能够找到两对匹配的元素）。

观察**while**循环，想一想如何能够使它的工作完成的更加精巧，如下所示：

```

while(it != v.end()) {
    cout << "adjacent_find: " << *it++
        << ", " << *it++ << endl;
    it = adjacent_find(it, v.end());
}

```

这个程序正是我们之前尝试的方式。但是该程序在任何编译器上都不可能得到期望的输出结果。这是因为对在循环内表达式中出现增1时的情况没有做出任何可靠的保证。

下一个测试使用了以**PlusOne**判定函数作为参数的**adjacent\_find()**，这个判定函数**PlusOne**能发现序列**v**中所有的下一个数比前一个数改变了1的元素位置。同样，采用**while**方法也能找到所有这样的情况。

算法**find\_first\_of()**需要另外一个对象范围来辅助，这由数组**b**提供。由于**find\_first\_of()**中的第1个和第2个范围由不同的模板参数控制，正如所见，这两个范围可以引用两个不同类型的容器。**find\_first\_of()**的第2种形式也进行了测试，使用了**PlusOne**。

**search()**算法精确地在第1个序列范围内找到了第2个范围序列，并且它们的元素具有相同

的顺序。**search()**的第2种形式使用了一个判定函数，该形式的典型应用是查找那些定义相等的序列，但也有可能进行更加有趣的查找——在这里，**PlusOne**判定函数找到的范围是{4,5,6}。

**find\_end()**测试发现了在整个序列的最后出现的{11,11,11}。为了显示它实际上已经找到了最后出现的这个子集，从迭代器**it**指向的位置开始打印**v**串的剩余部分。

第1个**search\_n()**测试寻找7的3个副本，找到它们并且打印出来。当使用**search\_n()**的第2种版本时，判定函数的出现一般意味着使用它来判定两个元素间的相等性，但是也可以有一些选择的自由。并且使用一个函数对象，这个函数对象是用15（在本例中）去乘序列中的值，并且检查它们是否大于100。也就是说，**search\_n()**检测要做的是“找到6连续的那些值，当被15乘时，每个产生的数大于100。”这不能精确地描述读者平常期望做的那些工作，但是却可以为下次遇到不寻常的查找问题时提供一些办法。

**min\_element()**和**max\_element()**算法很直观，但是看上去有些怪异。函数似乎是用一个“\*”来引用。实际上，返回的迭代器被释放掉以便产生打印的值。

为了测试替换，首先使用**replace\_copy()**（这样不会修改原始**vector**）以值47来替换所有值为8的元素。注意，用一个空的**vector v2**对**back\_inserter()**进行调用。为了演示**replace\_if()**，用标准模板**greater\_equal**连同**bind2nd**创建一个函数对象，用-1替换所有值大于等于7的元素。

### 6.3.6 比较范围

下面这些算法提供比较两个范围的方法。乍看起来，这些算法执行的运算类似**search()**函数。可是，**search()**查找的是第2个序列出现在第1个序列中的位置，而**equal()**和**lexicographical\_compare()**所做的只是进行两个序列的比较。另一方面，**mismatch()**比较两个序列在哪里停止同步比较，这两个序列必须有完全相同的长度。

```
bool equal(InputIterator first1, InputIterator last1,
           InputIterator first2);
bool equal(InputIterator first1, InputIterator last1,
           InputIterator first2, BinaryPredicate binary_pred);
```

在这两个函数中，**[first1,last1)**表示的第1个范围是一个典型的表示方法。第2个范围开始于**first2**，但是没有“**last2**”因为第2个范围的长度由第1个范围的长度来决定。如果两个范围完全相同（有相同的元素和相同的顺序），**equal()**函数返回真。在第1种情况中，由**operator==**执行比较，在第2种情况中，由**binary\_pred**来决定两个元素是否相同。

```
bool lexicographical_compare(InputIterator1 first1,
                             InputIterator1 last1, InputIterator2 first2,
                             InputIterator2 last2);
bool lexicographical_compare(InputIterator1 first1,
                             InputIterator1 last1, InputIterator2 first2,
                             InputIterator2 last2, BinaryPredicate binary_pred);
```

这两个函数决定第1个范围是否“字典编纂顺序的小于（lexicographically less）”第2个范围。（如果范围1小于范围2返回**true**，否则返回**false**。）字典编纂顺序的比较（lexicographical comparison）或称为“字典(dictionary)”比较，意味着比较的顺序等同于按字典规则建立字符串的顺序：每次比较一个元素。如果第1个元素不同，第1个元素就决定了两个字符串比较的结果，但是如果相同，算法移到下一个元素继续检查它们，如此下去直到遇到不匹配的那对元素为止。在这个点上，检查这对元素，如果范围1序列中的这个元素小于范围2序列中的相应元素，**lexicographical\_compare()**返回**true**；否则返回**false**。如果用能得到的所有方法从头

至尾扫描一个范围或另一个范围（本算法中范围的长度可以不同），都没有发现不相等的地方，范围1不小于范围2，因此函数返回**false**。

如果两个范围的长度不同，按字典编纂顺序，一个范围内缺少的元素起到“领先于(precede)”另一个范围内存在的元素的作用，因此“abc”领先于“abcd”。如果算法执行到一个范围的结尾，还没有找到不匹配的元素对，这时短的范围领先（按字典编纂顺序领先，即小）。在这种情况下，如果短的范围是第1个范围，则结果是**true**，反之是**false**。

在函数的第1种版本中，由**operator<**执行比较，在第2种版本中，使用判定函数**binary\_pred**。

```
pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1,
         InputIterator2 first2);
pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1,
         InputIterator2 first2, BinaryPredicate binary_pred);
```

这些算法如同在**equal()**中一样，进行比较的两个范围的长度完全相同，因此仅需要第2个范围的**first**迭代器，第1个范围的长度可以用来做第2个范围的长度。这个函数的功能与**equal()**正好相反，**equal()**仅比较两个范围是否相同，**mismatch()**告诉比较从哪里开始不同。为了完成这一工作，必须知道以下几点（1）第1个范围内出现不匹配的元素的位置；（2）第2个范围内出现不匹配的元素的位置。将两个迭代器一起装入一个**pair**对象并返回。如果没有出现不匹配，返回值是与第2个范围结合在一起的超越末尾的迭代器**last1**。**pair**模板类是一个**struct**，该**struct**含有两个用成员名**first**和**second**表示的元素，在<utility>头文件中定义。

如同在**equal()**中一样，第1个函数测试相等性使用**operator==**，而第2个使用**binary\_pred**。

#### 程序举例

因为标准C++ **string**类构造得如同一个容器（它含产生类型**string::iterator**的对象成员函数**begin()**和**end()**），可以方便地用来创建字符范围序列来测试STL比较算法。然而，需要注意的是，**string**有一个相当完整的属于自己的运算集，因此在使用STL算法执行运算之前，需要查看一下**string**类。

```
//: C06:Comparison.cpp
// The STL range comparison algorithms.
#include <algorithm>
#include <functional>
#include <string>
#include <vector>
#include "PrintSequence.h"
using namespace std;

int main() {
    // Strings provide a convenient way to create
    // ranges of characters, but you should
    // normally look for native string operations:
    string s1("This is a test");
    string s2("This is a Test");
    cout << "s1: " << s1 << endl << "s2: " << s2 << endl;
    cout << "compare s1 & s1: "
        << equal(s1.begin(), s1.end(), s1.begin()) << endl;
    cout << "compare s1 & s2: "
        << equal(s1.begin(), s1.end(), s2.begin()) << endl;
    cout << "lexicographical_compare s1 & s1: "
```



```

        << lexicographical_compare(s1.begin(), s1.end(),
                                   s1.begin(), s1.end()) << endl;
    cout << "lexicographical_compare s1 & s2: "
        << lexicographical_compare(s1.begin(), s1.end(),
                                   s2.begin(), s2.end()) << endl;
    cout << "lexicographical_compare s2 & s1: "
        << lexicographical_compare(s2.begin(), s2.end(),
                                   s1.begin(), s1.end()) << endl;
    cout << "lexicographical_compare shortened "
        << "s1 & full-length s2: " << endl;
    string s3(s1);
    while(s3.length() != 0) {
        bool result = lexicographical_compare(
            s3.begin(), s3.end(), s2.begin(), s2.end());
        cout << s3 << endl << s2 << ", result = "
            << result << endl;
        if(result == true) break;
        s3 = s3.substr(0, s3.length() - 1);
    }
    pair<string::iterator, string::iterator> p =
        mismatch(s1.begin(), s1.end(), s2.begin());
    print(p.first, s1.end(), "p.first", "");
    print(p.second, s2.end(), "p.second", "");
} ///:~

```

注意，**s1**和**s2**的惟一不同点是：**s2**的“Test”中的大写字母“T”。比较**s1**和**s2**的相等性产生**true**。不出所料，因为大写字母“T”，**s1**和**s2**不相等。

为了理解**lexicographical\_compare()**测试的输出结果，要记住两件事：首先，比较是按一个字母接着一个字母的顺序执行的；第二，现在C++编译系统的操作平台上，大写字母字符“领先于”小写字母字符。在第1个测试中，是**s1**与**s1**进行比较。这当然是完全相等。以字典编纂顺序进行比较，不会产生一个序列小于另外一个序列的结果（这是比较要寻找的结果），因此结果是**false**。第2个测试是问“**s1**领先于**s2**吗”？当比较进行到“test”中的第1个字符‘t’时，发现**s1**中的小写字母字符‘t’“大于”**s2**中的大写字母字符“T”，所以答案是**false**。但是，如果测试要看看**s2**是否领先于**s1**，答案是**true**。

为了更进一步地检测字典编纂顺序比较，本例中的下一个测试再次比较**s1**和**s2**（前面的比较返回**false**）。这次重复这个比较，每次通过循环减去**s1**（首先将**s1**复制到**s3**）末尾的一个字符，直到测试结果返回**true**。读者将会看到些什么？看到的是：只要从**s3**（**s1**的副本）中依次减到大写字母“T”，此时，则两个序列从开始一直到这一点都完全相等，不再进行计算。因为**s3**比**s2**短，**s3**字典编纂顺序领先于**s2**。

最后的测试使用**mismatch()**。为了得到返回值，现在创建合适的**pair p**，用第1个范围的迭代器类型及第2个范围的迭代器类型（在本例中，都是**string::iterator**）构造模板。为了打印该函数产生的结果，函数中第1个范围的不匹配迭代器是**p.first**，第2个范围的迭代器是**p.second**。在这两种情况中，从函数不匹配的迭代器开始到范围的末尾来打印该范围序列，所以可以准确地看到哪里是迭代器指出的点。

### 6.3.7 删除元素

因为STL的通用性，这里对删除的概念有一点限制。既然在STL中仅能通过迭代器“删除”元素，而迭代器可以指向数组、**vector**、**list**等，那么试图销毁正在被删除的元素和改变输入范围**[first,last)**的大小是不安全或是不合理的。（例如，一个已存在数组不能改变它的大小。）因此取而代之，STL“删除”函数重新排列该序列，就是将“已被删除的”元素排在序列的末

尾,“未删除的”元素排在序列的开头(与以前的顺序相同,只是减去被删除的元素——也就是说,这是一个稳定的操作)。然后函数返回一个指向序列的“新末尾”元素的迭代器,这个“新末尾”元素是不含被删除元素的序列的末尾,也是被删除元素序列的开头。换句话说,如果`new_last`是从“删除”函数返回的迭代器,则范围 `[first,new_last)` 是不包含任何被删除元素的序列,而范围 `[new_last, last)` 是被删除元素组成的序列。

如果想通过更多的STL算法来简单地使用序列,并把那些已被删除的元素包括在序列内,可以仅用`new_last`作为新的超越末尾的迭代器。但是,如果使用一个可以调整大小的容器`c`(不是一个数组),当想从容器中消除被删除的元素时,可以使用`erase()`来完成,例如:

```
c.erase(remove(c.begin(), c.end(), value), c.end());
```

也可以使用属于所有标准序列容器的`resize()`成员函数(更多关于此问题的内容将在第7章介绍)。

`remove()`的返回值是称为`new_last`的迭代器,而`erase()`则从`c`中真正删除掉所有的要被删除的元素。

`[new_last,last)`中的迭代器是能解析的,但是那些元素值未被指定,应该不再使用。

```
ForwardIterator remove(ForwardIterator first,
    ForwardIterator last, const T& value);
ForwardIterator remove_if(ForwardIterator first,
    ForwardIterator last, Predicate pred);
OutputIterator remove_copy(InputIterator first,
    InputIterator last, OutputIterator result, const T&
    value);
OutputIterator remove_copy_if(InputIterator first,
    InputIterator last, OutputIterator result, Predicate
    pred);
```

这里介绍的每一种“删除”形式都从头至尾遍历范围 `[first, last)`,找到符合删除标准的值,并且复制未被删除的元素覆盖已被删除的元素(因此可有效地删除元素)。未被删除的元素的原始排列顺序仍然保持。返回值是指向超越范围末尾的迭代器,该范围不包含任何已被删除的元素。这个迭代器指向的元素的值未被指定。

“if”版本的删除把每一个元素传递给判定函数`pred()`,来决定是否应该删除。(如果`pred()`返回`true`,则删除该元素。)“copy”版本的删除不需要修改原始序列,而取而代之是复制未被删除的值到一个开始于`result`的新范围,并返回指向新范围的超越末尾的迭代器。

```
ForwardIterator unique(ForwardIterator first,
    ForwardIterator last);
ForwardIterator unique(ForwardIterator first,
    ForwardIterator last, BinaryPredicate binary_pred);
OutputIterator unique_copy(InputIterator first,
    InputIterator last, OutputIterator result);
OutputIterator unique_copy(InputIterator first,
    InputIterator last, OutputIterator result,
    BinaryPredicate binary_pred);
```

在这些算法中,“unique”函数的每一种版本都从头至尾遍历范围 `[first, last)`,找到相邻的相等值(即副本),并且通过复制覆盖它们来“删除”这些副本。未被删除的元素的原始顺序仍然保持不变。返回值是指向该范围的超越末尾的迭代器,该范围相邻副本已被删除。

因为要删除的只是相邻的副本,因此如果有可能的话,在调用“unique”算法之前,调用

**sort()**，这样就能保证全部的副本都被删除掉。

对于输入范围内的每个迭代器的值*i*，包含在**binary\_pred**调用版本中：

```
binary_pred(*i, *(i-1));
```

如果返回值是**true**，则认为*\*i*是一个副本。

“copy”版本不改变原始序列，取而代之复制未被删除的值到一个开始于**result**的新范围，并返回指向新范围的超越末尾的迭代器。

#### 程序举例

这个例子给出了“remove”和“unique”函数工作的一个演示。

```
//: C06:Removing.cpp
// The removing algorithms.
//{L} Generators
#include <algorithm>
#include <cctype>
#include <string>
#include "Generators.h"
#include "PrintSequence.h"
using namespace std;

struct IsUpper {
    bool operator()(char c) { return isupper(c); }
};

int main() {
    string v;
    v.resize(25);
    generate(v.begin(), v.end(), CharGen());
    print(v.begin(), v.end(), "v original", "");
    // Create a set of the characters in v:
    string us(v.begin(), v.end());
    sort(us.begin(), us.end());
    string::iterator it = us.begin(), cit = v.end(),
        uend = unique(us.begin(), us.end());
    // Step through and remove everything:
    while(it != uend) {
        cit = remove(v.begin(), cit, *it);
        print(v.begin(), v.end(), "Complete v", "");
        print(v.begin(), cit, "Pseudo v ", " ");
        cout << "Removed element:\t" << *it
              << "\nPseudo Last Element:\t"
              << *cit << endl << endl;
        ++it;
    }
    generate(v.begin(), v.end(), CharGen());
    print(v.begin(), v.end(), "v", "");
    cit = remove_if(v.begin(), v.end(), IsUpper());
    print(v.begin(), cit, "v after remove_if IsUpper", " ");
    // Copying versions are not shown for remove()
    // and remove_if().
    sort(v.begin(), cit);
    print(v.begin(), cit, "sorted", " ");
    string v2;
    v2.resize(cit - v.begin());
    unique_copy(v.begin(), cit, v2.begin());
    print(v2.begin(), v2.end(), "unique_copy", " ");
    // Same behavior:
    cit = unique(v.begin(), cit, equal_to<char>());
    print(v.begin(), cit, "unique equal_to<char>", " ");
} //:~
```



字符串**v**是一个由随机产生的字符填满的字符容器。每个字符在**remove**语句中都被使用，但是每次都显示全部的字符串**v**，因此在得到结束点以后（存储在**cit**中），就可以看到该范围的剩余部分到底发生了什么变化。

为了演示**remove\_if()**，在函数对象类**IsUpper**中调用标准C库函数**isupper()**（在**<cctype>**中），将一个对象作为一个判定函数传递给**remove\_if()**。仅当字符是大写的时候返回**true**，因此只保留小写字符。在这里，在**print()**的调用中使用了范围的末尾作为参数，因此仅显示保留的元素。**remove()**和**remove\_if()**的复制形式没有演示，因为它们是非复制版本的一个简单变种，无需例子就应该会使用。

先对小写字母的序列进行排序，为测试“**unique**”函数做准备。（如果该序列未排序，则“**unique**”函数就不能被定义，但这大概并不是读者想要的。）首先，**unique\_copy()**使用默认的元素比较将序列中独一无二的元素放入一个新的**vector**中，然后再使用含有判定函数的**unique()**形式。判定函数嵌入到函数对象**equal\_to()**中，它与默认的元素比较产生相同的结果。

### 6.3.8 对已排序的序列进行排序和运算

STL算法的一个重要种类就是必须对已排好序的范围序列进行运算。STL提供了大量独立的排序算法，分别对应于稳定的、部分的或仅是规则的（不稳定的）排序。说也奇怪，只有部分排序有复制的版本。如果使用其他排序算法并且需要在一个副本上工作，那么就需要在排序前由用户自己来完成复制工作。

对于一个已经排好序的序列，可以在该序列上执行多种运算，包括从该序列中找出指定的某个元素或某组元素，到与另外的一个已排序的序列进行合并，或像数学集合一样来运算该序列等等。

对已排好序的序列进行包括排序或运算的每个算法都有两种版本。第1种版本使用对象自己的**operator<**来执行比较，第2种版本用**operator()(a,b)**来决定**a**和**b**的相对顺序。除此之外没有什么不同之处，所以不会在每个算法的描述中都指出这个不同点。

#### 1. 排序

排序算法需要由随机存取的迭代器来限制序列的范围，比如**vector**或**deque**。**list**容器有自己的嵌入**sort()**函数，因为它仅提供双向的迭代。

```
void sort(RandomAccessIterator first, RandomAccessIterator
    last);
void sort(RandomAccessIterator first, RandomAccessIterator
    last, StrictWeakOrdering binary_pred);
```

这些算法将**[first,last)**范围内的序列按升序顺序排序。第1种形式使用**operator<**，第2种形式使用提供的比较器对象来决定顺序。

```
void stable_sort(RandomAccessIterator first,
    RandomAccessIterator last);
void stable_sort(RandomAccessIterator first,
    RandomAccessIterator last, StrictWeakOrdering
    binary_pred);
```

这些算法将**[first,last)**范围内的序列按升序顺序排序，保持相等元素的原始顺序。（假设元素可以是相等的但不是相同的，这一点很重要。）

```
void partial_sort(RandomAccessIterator first,
    RandomAccessIterator middle, RandomAccessIterator last);
```

```
void partial_sort(RandomAccessIterator first,
    RandomAccessIterator middle, RandomAccessIterator last,
    StrictWeakOrdering binary_pred);
```

这些算法对来自**[first,last)**范围中的一些数量的元素进行排序，这些元素可以放入范围**[first,middle)**中。排序结束，在范围**[middle,last)**中其余的那些元素并不保证它们的顺序。

```
RandomAccessIterator partial_sort_copy(InputIterator first,
    InputIterator last, RandomAccessIterator result_first,
    RandomAccessIterator result_last);
RandomAccessIterator partial_sort_copy(InputIterator first,
    InputIterator last, RandomAccessIterator result_first,
    RandomAccessIterator result_last, StrictWeakOrdering
    binary_pred);
```

这些算法对来自**[first,last)**范围中的一些数量的元素进行排序，这些元素可以放入范围**[result\_first,result\_last)**中，并且复制这些元素到**[result\_first,result\_last)**。如果范围**[first,last)**比**[result\_first,result\_last)**小，则使用较少的元素。

```
void nth_element(RandomAccessIterator first,
    RandomAccessIterator nth, RandomAccessIterator last);
void nth_element(RandomAccessIterator first,
    RandomAccessIterator nth, RandomAccessIterator last,
    StrictWeakOrdering binary_pred);
```

这些算法如同**partial\_sort()**、**nth\_element()**部分地处理（排列）范围内的元素。但是，它比**partial\_sort()**要“少处理”得多。**nth\_element()**惟一保证的是无论选择什么位置，该位置都会成为一个分界点。范围**[first,nth)**内的所有元素都会成对地满足二元判定函数（通常默认的是**operator<**），而范围**(nth,last]**内的所有元素都不满足该判定。但是，任何一个子范围都不会是一个以特定的顺序排好序的序列，这不像**partial\_sort()**，它的第1个范围已排好序。

如果需要的是很弱的排序处理（例如，决定中值、百分点等等），这个算法要比**partial\_sort()**快得多。

## 2. 在已排序的范围中找出指定元素

一旦某个范围被排好序，就可以在范围内使用一系列运算来查找元素。在下面的函数中，总是存在有两种形式。一种是假定内在的**operator<**来执行排序，第2种运算符是使用一些其他的比较函数对象来执行排序。必须使用与执行排序相同的比较方法来定位元素；否则，结果不确定。另外，如果试图在未排序的范围上使用这些函数，结果将不可预料。

```
bool binary_search(ForwardIterator first, ForwardIterator
    last, const T& value);
bool binary_search(ForwardIterator first, ForwardIterator
    last, const T& value, StrictWeakOrdering binary_pred);
```

这些算法告诉用户是否**value**出现在已排序的范围**[first,last)**中。

```
ForwardIterator lower_bound(ForwardIterator first,
    ForwardIterator last, const T& value);
ForwardIterator lower_bound(ForwardIterator first,
    ForwardIterator last, const T& value, StrictWeakOrdering
    binary_pred);
```

这些算法返回一个迭代器，该迭代器指出**value**在已排序的范围**[first,last)**中第1次出

现的位置。如果**value**没有出现，返回的迭代器则指出它在该序列中应该出现的位置。

```
ForwardIterator upper_bound(ForwardIterator first,
    ForwardIterator last, const T& value);
ForwardIterator upper_bound(ForwardIterator first,
    ForwardIterator last, const T& value, StrictWeakOrdering
    binary_pred);
```

这些算法返回一个迭代器，该迭代器指出在已排序的范围 **[first,last)** 中超越**value**最后出现的一个位置。如果**value**没有出现，返回的迭代器则指出它在该序列中应该出现的位置。

```
pair<ForwardIterator, ForwardIterator>
equal_range(ForwardIterator first, ForwardIterator last,
    const T& value);
pair<ForwardIterator, ForwardIterator>
equal_range(ForwardIterator first, ForwardIterator last,
    const T& value, StrictWeakOrdering binary_pred);
```

在这些算法中，本质上结合了**lower\_bound( )**和**upper\_bound( )**，返回一个指出**value**在已排序的范围 **[first,last)** 中的首次出现和超越最后出现的**pair**。如果没有找到，这两个迭代器都指出**value**在该序列中应该出现的位置。

读者可能会惊讶于一个发现，即二分查找（也称折半查找）算法使用一个前向顺序查找的迭代器而不是随机存取的迭代器。（绝大多数对二分查找的解释是使用索引。）记住随机存取迭代器“是（is-a）”向前顺序查找的迭代器，它可以用在后者（向前顺序查找的）指定的地方。如果传递给这些算法之一的迭代器实际上支持随机存取，则使用了有效率的对数时间查找，否则执行的是线性查找。<sup>①</sup>

### 3. 程序举例

下面的例子将输入的每一个单词转化成**NString**，并且将其加入到**vector<NString>**。然后使用**vector**来演示各种排序和查找算法。

```
//: C06:SortedSearchTest.cpp
// Test searching in sorted ranges.
// NString
#include <algorithm>
#include <cassert>
#include <ctime>
#include <cstdlib>
#include <cstdint>
#include <fstream>
#include <iostream>
#include <iterator>
#include <vector>
#include "NString.h"
#include "PrintSequence.h"
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
    typedef vector<NString>::iterator sit;
    char* fname = "Test.txt";
    if(argc > 1) fname = argv[1];
    ifstream in(fname);
    assure(in, fname);
```

① 通过读取**tag**，算法能够决定迭代器的类型，这将在第7章讨论。



```

srand(time(0));
cout.setf(ios::boolalpha);
vector<NString> original;
copy(istream_iterator<string>(in),
    istream_iterator<string>(), back_inserter(original));
require(original.size() >= 4, "Must have four elements");
vector<NString> v(original.begin(), original.end()),
    w(original.size() / 2);
sort(v.begin(), v.end());
print(v.begin(), v.end(), "sort");
v = original;
stable_sort(v.begin(), v.end());
print(v.begin(), v.end(), "stable_sort");
v = original;
sit it = v.begin(), it2;
// Move iterator to middle
for(size_t i = 0; i < v.size() / 2; i++)
    ++it;
partial_sort(v.begin(), it, v.end());
cout << "middle = " << *it << endl;
print(v.begin(), v.end(), "partial_sort");
v = original;
// Move iterator to a quarter position
it = v.begin();
for(size_t i = 0; i < v.size() / 4; i++)
    ++it;
// Less elements to copy from than to the destination
partial_sort_copy(v.begin(), it, w.begin(), w.end());
print(w.begin(), w.end(), "partial_sort_copy");
// Not enough room in destination
partial_sort_copy(v.begin(), v.end(), w.begin(), w.end());
print(w.begin(), w.end(), "w partial_sort_copy");
// v remains the same through all this process
assert(v == original);
nth_element(v.begin(), it, v.end());
cout << "The nth_element = " << *it << endl;
print(v.begin(), v.end(), "nth_element");
string f = original[rand() % original.size()];
cout << "binary search: "
    << binary_search(v.begin(), v.end(), f) << endl;
sort(v.begin(), v.end());
it = lower_bound(v.begin(), v.end(), f);
it2 = upper_bound(v.begin(), v.end(), f);
print(it, it2, "found range");
pair<sit, sit> ip = equal_range(v.begin(), v.end(), f);
print(ip.first, ip.second, "equal_range");
} ///:~

```

这个例子使用前面见过的**NString**类，它存储一个字符串的副本出现的次数。**stable\_sort()**的调用显示了含相等字符串的对象的原始顺序是如何保存的。同时也可以看到在“部分排序”期间到底发生什么事情（保留的未排序的元素处在非特定的顺序之中）。不存在“部分的稳定排序。”

注意，在**nth\_element()**的调用中，无论nth元素变成什么（因为**URandGen**发生器，它可以从一个元素变成另一个元素），其前面的元素总是小于它，后面的元素大于它，此外这些元素并没有特定的顺序。由于**URandGen**发生器不存在副本，但是如果使用允许副本的发生器，将会看到nth元素以前的元素小于等于该nth元素。

这个例子也演示了全部的3个二分查找算法。同介绍过的一样，**lower\_bound()**用来查找序列中第1个等于给定关键字值的元素，**upper\_bound()**指向个最后一个符合条件元素的

下一元素，而`equal_range()`将两个结果作为一对数据返回。

#### 4. 合并已排序的序列

如同前面一样，每个函数的第1种形式假定由内在的`operator<`执行排序。第2种形式必须使用一些其他比较函数对象执行排序。必须使用与执行排序相同的比较方法来定位元素；否则，结果不确定。另外，如果试图在未排序的序列范围上使用这些算法，结果也会不可预料。

```
OutputIterator merge(InputIterator1 first1, InputIterator1
    last1, InputIterator2 first2, InputIterator2 last2,
    OutputIterator result);
OutputIterator merge(InputIterator1 first1, InputIterator1
    last1, InputIterator2 first2, InputIterator2 last2,
    OutputIterator result, StrictWeakOrdering binary_pred);
```

在这些算法中，从`[first1,last1)`和`[first2,last2)`中复制元素到`result`，这样在结果范围的序列以升序的顺序排序。这是一个稳定的运算。

```
void inplace_merge(BidirectionalIterator first,
    BidirectionalIterator middle, BidirectionalIterator
    last);
void inplace_merge(BidirectionalIterator first,
    BidirectionalIterator middle, BidirectionalIterator last,
    StrictWeakOrdering binary_pred);
```

这里假定`[first,middle)`和`[middle,last)`是在相同的序列中已排好序的两个范围。合并这两个范围序列到一个结果序列，该结果序列范围`[first,last)`包含将两个排好序的范围结合成一个有序的范围。

#### 5. 程序举例

很容易看到，如果在合并中使用`int`类型将会发生什么事情。下面的例子同时也强调了算法（以及我们自己定义的`print`模板）是怎样与数组和容器一起工作的：

```
//: C06:MergeTest.cpp
// Test merging in sorted ranges.
//{L} Generators
#include <algorithm>
#include "PrintSequence.h"
#include "Generators.h"
using namespace std;

int main() {
    const int SZ = 15;
    int a[SZ*2] = {0};
    // Both ranges go in the same array:
    generate(a, a + SZ, SkipGen(0, 2));
    a[3] = 4;
    a[4] = 4;
    generate(a + SZ, a + SZ*2, SkipGen(1, 3));
    print(a, a + SZ, "range1", " ");
    print(a + SZ, a + SZ*2, "range2", " ");
    int b[SZ*2] = {0}; // Initialize all to zero
    merge(a, a + SZ, a + SZ, a + SZ*2, b);
    print(b, b + SZ*2, "merge", " ");
    // Reset b
    for(int i = 0; i < SZ*2; i++)
        b[i] = 0;
    inplace_merge(a, a + SZ, a + SZ*2);
    print(a, a + SZ*2, "inplace_merge", " ");
    int* end = set_union(a, a + SZ, a + SZ, a + SZ*2, b);
```



```
    print(b, end, "set_union", " ");
} ///:~
```

在`main()`中，不是创建两个独立的数组，而是在数组**a**中创建两个首尾相连的范围。（这为`inplace_merge`带来方便。）第1个`merge()`的调用把结果放入一个不同的数组**b**中。为了进行比较，同时也调用`set_union()`，它与第1个`merge()`的调用有相同的标识符及类似的行为，除了它从第2个集合中删除副本。最后，`inplace_merge()`将**a**的两个部分结合到一起。

#### 6. 在已排序的序列上进行集合运算

一旦范围已排好序，就可以在其上执行数学集合运算。

```
bool includes(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2);
bool includes(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2,
              StrictWeakOrdering binary_pred);
```

在这些算法中，如果`[first2,last2)`是`[first1,last1)`的一个子集，返回`true`。没有任何一个范围要求只持有与另一个范围完全不同的元素，但是如果`[first2,last2)`持有`n`个特定值的元素，假如要想使返回结果为`true`，`[first1,last1)`也必须同时至少持有`n`个元素。

```
OutputIterator set_union(InputIterator1 first1,
                        InputIterator1 last1, InputIterator2 first2,
                        InputIterator2 last2, OutputIterator result);
OutputIterator set_union(InputIterator1 first1,
                        InputIterator1 last1, InputIterator2 first2,
                        InputIterator2 last2, OutputIterator result,
                        StrictWeakOrdering binary_pred);
```

这些算法在`result`范围中创建两个已排序范围的数学并集，返回值指向输出范围的末尾。没有任何一个输入范围要求只持有与另一个范围完全不同的元素，但是，如果在两个输入集合中多次出现某个特定值，结果集合中将包含完全相同的值出现的较大次数。

```
OutputIterator set_intersection(InputIterator1 first1,
                               InputIterator1 last1, InputIterator2 first2,
                               InputIterator2 last2, OutputIterator result);
OutputIterator set_intersection(InputIterator1 first1,
                               InputIterator1 last1, InputIterator2 first2,
                               InputIterator2 last2, OutputIterator result,
                               StrictWeakOrdering binary_pred);
```

这些算法在`result`中产生两个输入集合的交集，返回值指向输出范围的末尾——即在两个输入集合中都出现的数值的集合。没有任何一个输入范围要求只持有与另一个范围完全不同的元素，但是如果某个特定值在两个输入集合中多次出现，结果集合中将包含完全相同的值出现的较小次数。

```
OutputIterator set_difference(InputIterator1 first1,
                             InputIterator1 last1, InputIterator2 first2,
                             InputIterator2 last2, OutputIterator result);
OutputIterator set_difference(InputIterator1 first1,
                             InputIterator1 last1, InputIterator2 first2,
                             InputIterator2 last2, OutputIterator result,
                             StrictWeakOrdering binary_pred);
```

这些算法在`result`中产生数学上集合的差，返回值指向输出结果范围的末尾。所有出现在

**[frist1, last1)** 中, 但不在**[first2, last2)** 中出现的元素都放入结果集合。没有任何一个输入范围要求只持有独特的元素, 但是如果某个特定值在两个输入集合中多次出现 (在集合1中 **n** 次, 集合2中 **m** 次), 结果集合将包含这个值的 **max(n-m, 0)** 个副本。

```
OutputIterator set_symmetric_difference(InputIterator1
    first1, InputIterator1 last1, InputIterator2 first2,
    InputIterator2 last2, OutputIterator result);
OutputIterator set_symmetric_difference(InputIterator1
    first1, InputIterator1 last1, InputIterator2 first2,
    InputIterator2 last2, OutputIterator result,
    StrictWeakOrdering binary_pred);
```

在**result**集合构成中, 包括:

- 1) 所有在集合1中而不在集合2中的元素。
- 2) 所有在集合2中而不在集合1中的元素。

在这些算法中, 没有任何一个输入范围要求只持有独特的元素, 但是如果某个特定值在两个输入集合中多次出现 (在集合1中 **n** 次, 集合2中 **m** 次), 结果集合将包含这个值的 **abs(n-m)** 个副本, 其中**abs()**是取绝对值函数。返回值指向输出结果范围的末尾。

#### 7. 程序举例

观察仅使用字符的**vector**来演示集合运算将更加容易。这些字符是随机产生的, 并被排序, 但保留了副本, 当有了副本时, 现在就可以看到集合运算怎样执行。

```
//: C06:SetOperations.cpp
// Set operations on sorted ranges.
//{L} Generators
#include <algorithm>
#include <vector>
#include "Generators.h"
#include "PrintSequence.h"
using namespace std;

int main() {
    const int SZ = 30;
    char v[SZ + 1], v2[SZ + 1];
    CharGen g;
    generate(v, v + SZ, g);
    generate(v2, v2 + SZ, g);
    sort(v, v + SZ);
    sort(v2, v2 + SZ);
    print(v, v + SZ, "v", "");
    print(v2, v2 + SZ, "v2", "");
    bool b = includes(v, v + SZ, v + SZ/2, v + SZ);
    cout.setf(ios::boolalpha);
    cout << "includes: " << b << endl;
    char v3[SZ*2 + 1], *end;
    end = set_union(v, v + SZ, v2, v2 + SZ, v3);
    print(v3, end, "set_union", "");
    end = set_intersection(v, v + SZ, v2, v2 + SZ, v3);
    print(v3, end, "set_intersection", "");
    end = set_difference(v, v + SZ, v2, v2 + SZ, v3);
    print(v3, end, "set_difference", "");
    end = set_symmetric_difference(v, v + SZ,
        v2, v2 + SZ, v3);
    print(v3, end, "set_symmetric_difference", "");
} ///:~
```

在**v**和**v2**产生、排序和打印之后, 通过观察**v**的全部范围是否包含**v**的后半部分来测试

**includes()** 算法。如果包括，结果通常应该是真。数组 **v3** 保存 **set\_union()**、**set\_intersection()**、**set\_difference()** 和 **set\_symmetric\_difference()** 的输出结果，每一个结果都显示出来，这样读者就可以分析、思考它们，并确信算法正如预想的那样执行。

### 6.3.9 堆运算

堆是一个像数组的数据结构，用来实现“优先队列”，“优先队列”是一个靠优先权调节检索元素的方式来组织的序列，其中优先权是依据某些比较函数决定的。标准库中的堆运算允许一个序列被视为是一个“堆”数据结构，这通常可以有效地返回最高优先权的元素，而无需全部排序整个序列。

如同“排序”运算一样，每个函数都有两种版本。第1种使用对象自己的 **operator<** 来执行比较；第2种使用另外的 **StrictWeakOrdering** 对象的 **operator()(a,b)** 来比较两个对象：**a < b**。

```
void make_heap(RandomAccessIterator first,
               RandomAccessIterator last);
void make_heap(RandomAccessIterator first,
               RandomAccessIterator last,
               StrictWeakOrdering binary_pred);
```

这些算法将一个任意序列范围转化成堆。

```
void push_heap(RandomAccessIterator first,
               RandomAccessIterator last);
void push_heap(RandomAccessIterator first,
               RandomAccessIterator last,
               StrictWeakOrdering binary_pred);
```

这些算法向由范围 **[first, last-1)** 决定的堆中增加元素 **\*(last-1)**。换句话说，将最后一个元素放入堆中合适的位置。

```
void pop_heap(RandomAccessIterator first,
              RandomAccessIterator last);
void pop_heap(RandomAccessIterator first,
              RandomAccessIterator last,
              StrictWeakOrdering binary_pred);
```

在这些算法中，将最大的元素（在运算前实际上在 **\*first** 中，这是堆定义方式的缘故）放入位置 **\*(last-1)** 并且重新组织剩余的范围，使其仍然在堆的顺序中。如果只是抓取 **\*first**，下一个元素就将不是下一个最大的元素；因此，如果想以完全优先队列的顺序保持堆，必须调用 **pop\_heap()** 来完成这个运算。

```
void sort_heap(RandomAccessIterator first,
               RandomAccessIterator last);
void sort_heap(RandomAccessIterator first,
               RandomAccessIterator last,
               StrictWeakOrdering binary_pred);
```

可以将这些算法完成的工作想象为 **make\_heap()** 的补充。它使一个以堆顺序排列的序列，转化成普通的排列顺序，这样它就不再是一个堆。这意味着如果调用 **sort\_heap()**，将不能再在这个序列范围上使用 **push\_heap()** 或 **pop\_heap()**。（当然，你可以使用这些函数，但不会完成任何有意义的工作。）这是个不稳定的排序。

### 6.3.10 对某一范围内的所有元素进行运算

这些算法遍历整个范围并对每个元素执行运算。它们在利用运算的结果方面有所不同：



**for\_each()** 丢弃运算的返回值，而 **transform()** 将每个运算的结果放入一个目的序列（也可以是原始序列）。

```
UnaryFunction for_each(InputIterator first, InputIterator
    last, UnaryFunction f);
```

在该算法中，对 **[first,last)** 中的每个元素应用函数对象 **f**，丢弃每个个别的 **f** 应用的返回值。如果 **f** 仅是一个函数指针，说明这是典型的与返回值无关；但是，如果 **f** 是一个保留某些内部状态的对象，则该对象可以捕获一个返回值，它们结合到一起应用到该范围上。**for\_each()** 的最终返回值是 **f**。

```
OutputIterator transform(InputIterator first, InputIterator
    last, OutputIterator result, UnaryFunction f);
OutputIterator transform(InputIterator1 first,
    InputIterator1 last, InputIterator2 first2,
    OutputIterator result, BinaryFunction f);
```

这些算法，如同 **for\_each()** 一样，**transform()** 对范围 **[first,last)** 中的每个元素应用函数对象 **f**。但是，不是丢弃每次函数调用的结果，**transform()** 而是将结果复制（使用 **operator=**）到 **\*result**，每次复制后增加 **result** 的内容。（**result** 指向的序列必须有足够的存储空间；否则，用一个插入符强迫插入来代替赋值。）

**transform()** 的第1种形式仅调用了 **f(\*first)**，在这里第1个范围表示一个输入序列。类似地，第2种形式调用 **f(\*first1,\*first2)**。（注意，第2个输入范围的长度由第1个输入范围的长度决定。）这两种情况的返回值都是超越末尾的迭代器，该迭代器指出结果输出范围。

#### 程序举例

因为对容器中的对象做的大部分工作是对所有这些对象应用某个运算，这些都是相当重要的算法，值得为此给出一些例证。

首先，分析 **for\_each()**。它扫描整个范围，依次提取每个元素并把它作为一个参数进行传递，如同调用的任何被授予的函数对象一样。因此，**for\_each()** 执行那些由用户编写的规范的运算。如果想在编译器的头文件中查看 **for\_each()** 的模板定义，将会看到下述编码：

```
template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last,
    Function f) {
    while(first != last)
        f(*first++);
    return f;
}
```

下面的例子显示了一些能够扩展这个模板的几种方法。首先，需要一个保持追踪它的对象的类，这样我们就可以知道这些对象被适当地销毁掉：

```
//: C06:Counted.h
// An object that keeps track of itself.
#ifdef COUNTED_H
#define COUNTED_H
#include <vector>
#include <iostream>

class Counted {
    static int count;
    char* ident;
public:
    Counted(char* id) : ident(id) { ++count; }
```



```

~Counted() {
    std::cout << ident << " count = "
               << --count << std::endl;
}
};

class CountedVector : public std::vector<Counted*> {
public:
    CountedVector(char* id) {
        for(int i = 0; i < 5; i++)
            push_back(new Counted(id));
    }
};
#endif // COUNTED_H ///:~

//: C06:Counted.cpp {0}
#include "Counted.h"
int Counted::count = 0;
///:~

```

**class Counted**对已创建的**Counted**对象的个数保存一个静态的计数，并且当这些对象被销毁时通知用户<sup>①</sup>。另外，每个**Counted**对象保存一个**char\***标识符以便追踪输出更加容易。

**CountedVector**由**vector<Counted\*>**派生而来，并且在构造函数中创建一些**Counted**对象，处理每个想要的**char\***。**CountedVector**使测试相当简单，如下所示：

```

//: C06:ForEach.cpp {-mwcc}
// Use of STL for_each() algorithm.
//{L} Counted
#include <algorithm>
#include <iostream>
#include "Counted.h"
using namespace std;

// Function object:
template<class T> class DeleteT {
public:
    void operator()(T* x) { delete x; }
};

// Template function:
template<class T> void wipe(T* x) { delete x; }

int main() {
    CountedVector B("two");
    for_each(B.begin(), B.end(), DeleteT<Counted>());
    CountedVector C("three");
    for_each(C.begin(), C.end(), wipe<Counted>());
} ///:~

```

显然，在这里有一些事情需要反复多次去做，既然这样，为什么不创建一个算法用**delete**来删除容器中所有的指针呢？可以使用**transform()**来完成这项工作。**transform()**优于**for\_each()**的地方在于**transform()**将调用函数对象的结果赋给结果范围，该结果范围实际上是输入范围。这种情况意味着对输入范围的序列进行逐字的转换，因为每个元素是原先值的一个修改。在本例中这个方法尤其有用，因为在对每个指针调用**delete**后，为其赋安全的零值更加适合。**transform()**可以很容易地做到这些：

<sup>①</sup> 在这个例子中我们忽略了拷贝构造函数和赋值操作，因为没有使用它们。

```

//: C06:Transform.cpp {-mwcc}
// Use of STL transform() algorithm.
//{L} Counted
#include <iostream>
#include <vector>
#include <algorithm>
#include "Counted.h"
using namespace std;

template<class T> T* deleteP(T* x) { delete x; return 0; }

template<class T> struct Deleter {
    T* operator()(T* x) { delete x; return 0; }
};

int main() {
    CountedVector cv("one");
    transform(cv.begin(), cv.end(), cv.begin(),
        deleteP<Counted>);
    CountedVector cv2("two");
    transform(cv2.begin(), cv2.end(), cv2.begin(),
        Deleter<Counted>());
} ///:~

```

这里显示了两种方法：使用模板函数或模板化的函数对象。在调用**transform()**后，**vector**包含5个空指针，它更安全因为用它对任何副本**delete**都是无效的。

有一件事不能做，那就是在遍历中**delete**每个指针，而没有在函数或对象内部封装对**delete**的调用。即如下面所做：

```
for_each(a.begin(), a.end(), ptr_fun(operator delete));
```

这与前面的**destroy()**调用有相同的问题：**operator delete()**获取一个**void\***，但是迭代器并不是指针。甚至如果要对它进行编译，得到的将是一系列用来释放存储空间的函数调用。不能得到对**a**中每个指针都调用**delete**的效果，然而不会调用析构函数。这显然不是想要的结果，所以需要封装对**delete**的调用。

在前面的**for\_each()**的例子中，忽略了算法的返回值。这个返回值是传递给**for\_each()**的函数。如果这个函数仅是指向一个函数的指针，该返回值并不是很有用，但如果它是一个函数对象那就完全不同了，这个函数对象可能含有内部的成员数据，可以使用这些成员数据来积累关于在**for\_each()**中见到的所有对象的信息。

例如，考虑一个简单的存货清单模型。每个**Inventory**对象包含它所代表的产品类型（在这里用单个的字符表示产品项目名称）、该产品的数量以及每种产品的价格。

```

//: C06:Inventory.h
#ifndef INVENTORY_H
#define INVENTORY_H
#include <iostream>
#include <cstdlib>
using std::rand;

class Inventory {
    char item;
    int quantity;
    int value;
public:
    Inventory(char it, int quant, int val)
        : item(it), quantity(quant), value(val) {}
    // Synthesized operator= & copy-constructor OK

```



```

char getItem() const { return item; }
int getQuantity() const { return quantity; }
void setQuantity(int q) { quantity = q; }
int getValue() const { return value; }
void setValue(int val) { value = val; }
friend std::ostream& operator<< (
    std::ostream& os, const Inventory& inv) {
    return os << inv.item << ": "
        << "quantity " << inv.quantity
        << ", value " << inv.value;
    }
};

// A generator:
struct InvenGen {
    Inventory operator()() {
        static char c = 'a';
        int q = rand() % 100;
        int v = rand() % 500;
        return Inventory(c++, q, v);
    }
};
#endif // INVENTORY_H ///:~

```

成员函数取得产品项目的名称，取得并确定相应的数量和价格。**operator<<**向**ostream**打印出**Inventory**对象。用一个发生器来创建这些对象，这些对象含有顺序标记的产品项目名及随机的数量和价格。

为了找出产品项目的总数及全部价值，可以使用含有总计数据成员的**for\_each()**来创建一个函数对象：

```

//: C06:CalcInventory.cpp
// More use of for_each().
#include <algorithm>
#include <ctime>
#include <vector>
#include "Inventory.h"
#include "PrintSequence.h"
using namespace std;

// To calculate inventory totals:
class InvAccum {
    int quantity;
    int value;
public:
    InvAccum() : quantity(0), value(0) {}
    void operator()(const Inventory& inv) {
        quantity += inv.getQuantity();
        value += inv.getQuantity() * inv.getValue();
    }
    friend ostream&
    operator<<(ostream& os, const InvAccum& ia) {
        return os << "total quantity: " << ia.quantity
            << ", total value: " << ia.value;
    }
};

int main() {
    vector<Inventory> vi;
    srand(time(0)); // Randomize
    generate_n(back_inserter(vi), 15, InvenGen());
}

```



```

    print(vi.begin(), vi.end(), "vi");
    InvAccum ia = for_each(vi.begin(), vi.end(), InvAccum());
    cout << ia << endl;
} ///:~

```

**InvAccum**的**operator()**有一个参数，这是**for\_each()**要求的。当**for\_each()**遍历某个范围时，获取该范围的每一个对象并将其传递给**InvAccum::operator()**，它执行计算并保存结果。在这个处理的最后，**for\_each()**返回**InvAccum**对象，并打印该**InvAccum**对象。

使用**for\_each()**可以对**Inventory**对象做很多事。例如，**for\_each()**可以方便地将所有产品的价格增加10%。但是读者会注意到**Inventory**对象没有办法改变**item**的值。设计**Inventory**的程序员认为这是一个好的主意。毕竟，为什么想要改变一个商品的名称？但是在市场上的交易已经决定了要将所有的产品名称改为大写，使得它们看上去与“新的、改进的”产品一样。他们已经做了调研并且决定用新的产品名称来进行促销（好了，为了市场上的交易总需要做一些事情……）。所以这里不使用**for\_each()**，而是使用**transform()**：

```

///: C06:TransformNames.cpp
// More use of transform().
#include <algorithm>
#include <cctype>
#include <ctime>
#include <vector>
#include "Inventory.h"
#include "PrintSequence.h"
using namespace std;

struct NewImproved {
    Inventory operator()(const Inventory& inv) {
        return Inventory(toupper(inv.getItem()),
            inv.getQuantity(), inv.getValue());
    }
};

int main() {
    vector<Inventory> vi;
    srand(time(0)); // Randomize
    generate_n(back_inserter(vi), 15, InvenGen());
    print(vi.begin(), vi.end(), "vi");
    transform(vi.begin(), vi.end(), vi.begin(), NewImproved());
    print(vi.begin(), vi.end(), "vi");
} ///:~

```

注意，结果范围与输入范围相同；即，在适当的位置执行转换。

现在假设销售部门需要产生一个特价清单，对每种商品有不同的折扣。原始的清单必须原样保留，并且需要产生任意数量的特价清单。销售部门将为每个新清单提供一个单独的折扣明细表。为了解决这个问题，这里使用**transform()**的第2种版本：

```

///: C06:SpecialList.cpp
// Using the second version of transform().
#include <algorithm>
#include <ctime>
#include <vector>
#include "Inventory.h"
#include "PrintSequence.h"
using namespace std;

struct Discounter {
    Inventory operator()(const Inventory& inv,

```

```

        float discount) {
            return Inventory(inv.getItem(), inv.getQuantity(),
                int(inv.getValue() * (1 - discount)));
        }
    };

    struct DiscGen {
        float operator()() {
            float r = float(rand() % 10);
            return r / 100.0;
        }
    };

    int main() {
        vector<Inventory> vi;
        srand(time(0)); // Randomize
        generate_n(back_inserter(vi), 15, InvenGen());
        print(vi.begin(), vi.end(), "vi");
        vector<float> disc;
        generate_n(back_inserter(disc), 15, DiscGen());
        print(disc.begin(), disc.end(), "Discounts:");
        vector<Inventory> discounted;
        transform(vi.begin(), vi.end(), disc.begin(),
            back_inserter(discounted), Discounter());
        print(discounted.begin(), discounted.end(), "discounted");
    } ///:~

```

给定一个**Inventory**对象和一个折扣比率，**Discounter**函数对象产生一个新的含折扣价格的**Inventory**对象。**DiscGen**函数对象仅产生随机的从1%到10%之间的折扣值用来进行测试。在**main( )**中创建两个**vector**，一个用于**Inventory**，一个用于折扣。将它们随同**Discounter**对象传递给**transform( )**，**transform( )**填充一个新的称为**discounted**的**vector<Inventory>**对象。

### 6.3.11 数值算法

这些算法都包含在头文件**<numeric>**中，因为它们主要用来执行数值计算。

```

T accumulate(InputIterator first, InputIterator last, T
    result);
T accumulate(InputIterator first, InputIterator last, T
    result, BinaryFunction f);

```

第1种形式是一般化的合计，对于由迭代器**i**指向的 **[first,last)** 中的每一个元素，执行运算**result=result+\*i**，在这里**result**是**T**类型。但是，第2种形式更普遍；它对于范围中从头至尾的每一个元素**\*i**应用函数**f(result,\*i)**。

注意**transform( )**的第2种形式和**accumulate( )**的第2种形式之间的相似之处。

```

T inner_product(InputIterator1 first1, InputIterator1
    last1, InputIterator2 first2, T init);
T inner_product(InputIterator1 first1, InputIterator1
    last1, InputIterator2 first2, T init, BinaryFunction1
    op1, BinaryFunction2 op2);

```

在这些算法中，计算两个范围 **[first1,last1)** 和 **[first2,first2+(last1-first1))** 的一个广义内积。用第1个序列中的元素乘以第2个序列中“平行的”元素并对其积进行累加来产生返回值。因此，如果有两个序列 **{1,1,2,2}** 和 **{1,2,3,4}**，内积是

$$(1*1) + (1*2) + (2*3) + (2*4)$$

返回结果为17。参数**init**是内积的初始值——可能是0也可能是任何值，这对于空的第1个序列

尤其重要，因为它是默认的返回值。第2个序列必须至少含有与第1个序列一样多的元素。

第2种形式对它的序列应用一对函数。函数`op1`用来代替加法，而函数`op2`用来代替乘法。因此，如果对上面的序列应用`inner_product()`的第2种版本，结果会是下面的这些运算：

```
init = op1(init, op2(1,1));
init = op1(init, op2(1,2));
init = op1(init, op2(2,3));
init = op1(init, op2(2,4));
```

所以，这与`transform()`相似，但用执行两个运算来代替一个运算。

```
OutputIterator partial_sum(InputIterator first,
    InputIterator last, OutputIterator result);
OutputIterator partial_sum(InputIterator first,
    InputIterator last, OutputIterator result,
    BinaryFunction op);
```

这些算法计算一个广义部分和。创建一个新的在`result`开始的序列。新序列中每个元素都是`[first,last)`范围中从第1个元素到当前选择的元素之间所有元素的累加和。例如，如果原始序列是`{1,1,2,2,3}`，产生的结果序列是`{1,1+1,1+1+2,1+1+2+2,1+1+2+2+3}`，即`{1,2,4,6,9}`。

在第2种版本中，使用二元函数`op`代替`+`运算符，取得累积到那个点的所有的“合计”，并且把它与新值结合起来。例如，对上面的序列使用`multiplies<int>()`（一种乘法`op`）作为对象，输出结果是`{1,1,2,4,12}`。注意，在输入/输出两个序列中，第1个输出结果值始终与第1个输入值相同。

返回值指向输出范围`[result,result+(last-first))`的末尾。

```
OutputIterator adjacent_difference(InputIterator first,
    InputIterator last, OutputIterator result);
OutputIterator adjacent_difference(InputIterator first,
    InputIterator last, OutputIterator result, BinaryFunction
    op);
```

这些算法计算全部范围`[first,last)`中的相邻元素的差。这意味着在新序列中，每个元素的值是原始序列中当前元素与前面的元素的差值（第1个值不变）。例如，如果原始序列是`{1,1,2,2,3}`，结果序列是`{1,1-1,2-1,2-2,3-2}`，即`{1,0,1,0,1}`。

第2种形式使用二元函数`op`代替`-`运算符执行“求差”。例如，如果对序列使用`multiplies<int>()`作为函数对象（即用“乘法”代替“减法”），输出结果是`{1,1,2,4,6}`。

返回值指向输出范围`[result,result+(last-first))`的末尾。

程序举例

这个程序在整型数组上测试`<numeric>`头文件中所有的算法的两种形式。读者将会注意到，在程序例子提供的函数或函数群的形式测试中，这些函数对象被使用的形式是一致的，而使用形式一致则产生相同的结果，因此结果是完全相同的。这里也演示了更加清晰的运算，该运算继续下去就是如何替换用户自己的运算。

```
//: C06:NumericTest.cpp
#include <algorithm>
#include <iostream>
#include <iterator>
#include <functional>
```

```

#include <numeric>
#include "PrintSequence.h"
using namespace std;

int main() {
    int a[] = { 1, 1, 2, 2, 3, 5, 7, 9, 11, 13 };
    const int ASZ = sizeof a / sizeof a[0];
    print(a, a + ASZ, "a", " ");
    int r = accumulate(a, a + ASZ, 0);
    cout << "accumulate 1: " << r << endl;
    // Should produce the same result:
    r = accumulate(a, a + ASZ, 0, plus<int>());
    cout << "accumulate 2: " << r << endl;
    int b[] = { 1, 2, 3, 4, 1, 2, 3, 4, 1, 2 };
    print(b, b + sizeof b / sizeof b[0], "b", " ");
    r = inner_product(a, a + ASZ, b, 0);
    cout << "inner_product 1: " << r << endl;
    // Should produce the same result:
    r = inner_product(a, a + ASZ, b, 0,
        plus<int>(), multiplies<int>());
    cout << "inner_product 2: " << r << endl;
    int* it = partial_sum(a, a + ASZ, b);
    print(b, it, "partial_sum 1", " ");
    // Should produce the same result:
    it = partial_sum(a, a + ASZ, b, plus<int>());
    print(b, it, "partial_sum 2", " ");
    it = adjacent_difference(a, a + ASZ, b);
    print(b, it, "adjacent_difference 1", " ");
    // Should produce the same result:
    it = adjacent_difference(a, a + ASZ, b, minus<int>());
    print(b, it, "adjacent_difference 2", " ");
} ///:~

```

注意，**inner\_product()**和**partial\_sum()**的返回值是结果序列的超越末尾的迭代器，因此作为**print()**函数中的第2个迭代器。

因为每个函数的第2种形式允许用户提供自己的函数对象，所以仅函数的第1种形式是纯“数值的”。读者可以用**inner\_product()**做很多能想得到的非直观数值的事情。

### 6.3.12 通用实用程序

最后，这里还有与其他的算法一起使用的一些基本工具；用户自己可能会，也可能不会直接使用这些工具。

```

(Templates in the <utility> header)
template<class T1, class T2> struct pair;
template<class T1, class T2> pair<T1, T2>
    make_pair(const T1&, const T2&);

```

在本章前面描述并使用过这些工具。**pair**是一个简单的将两个对象（可能不同类型的对象）封装成一个对象的方法。当需要从一个函数返回多个对象时使用它是很典型的情况，但是也可以用来创建一个持有**pair**对象的容器，或将多个对象作为一个参数进行传递。通过**p.first**和**p.second**来访问指定的元素，这里**p**是**pair**对象。例如，本章中描述的**equal\_range()**函数，作为迭代器的**pair**来返回结果。可以直接**insert()**一个**pair**到**map**或**multimap**中；对于这些容器来说**pair**是**value\_type**。

如果想“在执行中”创建一个**pair**，典型的方法是使用模板函数**make\_pair()**，而不是显式地构造一个**pair**对象。**make\_pair()**会自动推断出它接收到的参数的类型，这样即减轻程序员打字负担，也增加了程序的健壮性。



```
(From <iterator>)
difference_type distance(InputIterator first, InputIterator
last);
```

该算法计算**first**与**last**之间的元素个数。更准确地说，它返回一个整数值，这个整数表示在**first**等于**last**之前它必须增加的次数。在这一处理过程中不会发生解析迭代器的现象。

```
(From<iterator>)
```

根据**n**的值前向移动迭代器**i**的位置。(如果迭代器是双向的,也可以根据**n**的负值向后移动。)这个算法意识到,对不同类型的迭代器应该采用不同的方法,而它使用的都是最有效的方法。例如,对随机迭代器可以用普通的算术(**i+=n**)直接增加,而双向迭代器必须增加**n**次。

```
(From <iterator>)
back_insert_iterator<Container>
    back_inserter(Container& x);
front_insert_iterator<Container>
    front_inserter(Container& x);
insert_iterator<Container>
    inserter(Container& x, Iterator i);
```

这些函数用来为给定的容器创建迭代器,以便向容器中插入元素,而不是用**operator=**覆盖容器中已存在的元素(这是默认的行为)。每种类型的迭代器对插入使用不同的运算:**back\_insert\_iterator**使用**push\_back()**,**front\_insert\_iterator**使用**push\_front()**,而**insert\_iterator**使用**insert()**(因此可以与关联式容器一起使用,而另外两种可以与顺序容器一起使用)。这些细节将在第7章中介绍。

```
const LessThanComparable& min(const LessThanComparable& a,
    const LessThanComparable& b);
const T& min(const T& a, const T& b,
    BinaryPredicate binary_pred);
```

在这些算法中,返回两个参数中较小的一个,或如果两个参数相等则返回第1个参数。第1种版本用**operator<**执行比较,而第2种版本将两个参数传递给**binary\_pred**来执行比较。

```
const LessThanComparable& max(const LessThanComparable& a,
    const LessThanComparable& b);
const T& max(const T& a, const T& b,
    BinaryPredicate binary_pred);
```

这些算法与**min()**很像,但是返回两个参数中较大的一个。

```
void swap(Assignable& a, Assignable& b);
void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
```

使用赋值的方法来交换**a**和**b**的值。注意,所有的容器类都使用特化的**swap()**版本,这比通用的版本要更有效得多。

**iter\_swap()**函数交换它涉及的两个参数的值。

## 6.4 创建自己的STL风格算法

只要适应了STL算法的风格,用户就可以开始创建自己的通用算法。因为这些算法符合STL中对所有其他算法的约定,使用自己编写的通用算法对熟悉STL的程序员来说更加容易,因此这也成为“扩展STL词汇表”的一种方式。

解决这个问题最容易的方法是在头文件**<algorithm>**中,找到那些与所需要的功能相似

的一些算法并将其作为样板，在其后模仿编写自己的代码。<sup>①</sup>（事实上，在头文件中所有STL实现都对模板直接提供代码。）

如果仔细观察标准C++库中算法的列表，就可能注意到一个明显的遗漏：没有`copy_if()`算法。尽管用`remove_copy_if()`可以完成相同的效果，但这样做相当不方便，因为必须要转化判定条件。（记住，`remove_copy_if()`仅复制那些不满足判定条件的元素，并有效地删除那些满足判定条件的元素。）

读者可能对用编写一个函数对象适配器来完成这项工作感兴趣，在将函数对象适配器传递给`remove_copy_if()`之前要取消掉那些不满足判定函数的元素，这意味着要通过如下的声明来完成：

```
// Assumes pred is the incoming condition
replace_copy_if(begin, end, not1(pred));
```

这看上去很合理，但是当读者想起使用判定函数时，而该判定函数是一个指向尚未完善的函数的指针，就会看到为什么它不能工作——`not1`期望的是一个能适应的函数对象，而现在不是这样。编写`copy_if()`的惟一解决方法是从零开始做起。既然从查阅其他复制算法中了解到对输入和输出需要两个单独的迭代器，那么就可以用下面的例子完成这一工作：

```
//: C06:copy_if.h
// Create your own STL-style algorithm.
#ifndef COPY_IF_H
#define COPY_IF_H

template<typename ForwardIter,
        typename OutputIter, typename UnaryPred>
OutputIter copy_if(ForwardIter begin, ForwardIter end,
                  OutputIter dest, UnaryPred f) {
    while(begin != end) {
        if(f(*begin))
            *dest++ = *begin;
        ++begin;
    }
    return dest;
}
#endif // COPY_IF_H ///:~
```

注意，`begin`的自增运算不能完整地进入到复制表达式之内。

## 6.5 小结

本章的目标是给读者一个关于标准模板库中算法实用性的理解。也就是说，使读者知道并能够轻松地了解STL，这样就可以在符合C++规则的基础上开始使用它（或者至少考虑使用它，这样一来，读者就会回到这里并寻找合适的解决方法。）STL是强大的，不仅因为它是合理且完全的工具库，而且因为它提供了考虑问题解决方案的词汇表，它也是创建附加工具的框架。

尽管本章给出了一些创建用户自己的工具的例子，但还没进入到完全理解STL的所有细微之处所必需的理论深度。一旦进入这样的理论深度，读者就会创建出比已经介绍过的例子更加复杂的工具。遗漏这些内容的部分原因是本教材篇幅的限制，但大部分原因是因为它已经超出了本教材对该章的要求——在这里，我们的目标是给读者一个实用性的理解，以便使读者一天一天地逐步改进自己的编程技巧。

<sup>①</sup> 当然，没有违反任何版权保护法律。

有大量的书籍专门讲解STL（在附录中列出了它们），但是作者在这里特别推荐Scott Meyers的《Effective STL》（Addison Wesley, 2002）。

## 6.6 练习

- 6-1 创建一个返回 **clock()**（在 **<ctime>** 头文件中）当前值的发生器。创建一个 **list<clock\_t>**，并且通过该发生器用 **generate\_n()** 填充它。在列表中删除任意副本，并且使用 **copy()** 把它打印到 **cout**。
- 6-2 使用 **transform()** 和 **toupper()**（在 **<cctype>** 头文件中），编写一个函数调用，将一个字符串全部转化成大写字母。
- 6-3 创建一个 **Sum** 函数对象模板，该函数对象模板在调用 **for\_each()** 时，累加范围内所有的值。
- 6-4 编写一个回文构词法发生器，以一个单词作为命令行参数，并且产生所有可能的字母排列。
- 6-5 编写一个“句子回文构词法发生器”，以一个句子作为命令行参数，并且产生所有可能的句子中单词的排列。（不要落下某些单词，仅是将它们前后左右移动。）
- 6-6 用基类 **B** 和派生类 **D** 创建一个类层次结构关系。在 **B** 中放入一个 **virtual** 成员函数 **void f()**，这样它将打印一个显示 **B** 中 **f()** 被调用的消息，且为 **D** 重新定义这个函数来打印一个不同的信息。创建一个 **vector<B\*>**，并且用 **B** 和 **D** 的对象填充它。使用 **for\_each()** 来为 **vector** 中的每个对象调用 **f()**。
- 6-7 修改 **FunctionObjects.cpp**，以便用 **float** 代替 **int**。
- 6-8 修改 **FunctionObjects.cpp**，模板化测试的主体，这样就能选择要测试的类型。（必须把 **main()** 的大部分放入到一个单独的模板函数中。）
- 6-9 编写一个程序，以一个整数作为命令行参数，并找出它的所有因数。
- 6-10 编写一个程序，以一个文本文件的名称作为命令行参数。打开这个文件并且每次读入一个单词（提示：使用 **>>**）。将每个词存储到一个 **vector<string>**。将所有的词转化成小写，存储它们，删除全部的副本并打印结果。
- 6-11 编写一个程序，使用 **set\_intersection()** 找出两个输入文件中共有的所有单词。修改程序，使用 **set\_symmetric\_difference()** 来显示两个输入文件中非共有的单词。
- 6-12 创建一个程序，在命令行给定一个整数，创建一个向上直到包括命令行数值在内的所有整数的阶乘的“阶乘表”。为了完成这个工作，编写一个发生器来填充 **vector<int>**，然后与标准函数对象一起使用 **partial\_sum()**。
- 6-13 修改 **CalcInventory.cpp**，使它能找到所有数量小于某个总数的对象。提供这个总数作为命令行参数，并使用 **copy\_if()** 和 **bind2nd()** 来创建小于目标值的数值的集合。
- 6-14 使用 **UrandGen()** 产生100个数。（数的大小没有关系。）找到范围中模23的同余数（意思是当被23除时有相同的余数）。读者手工挑选一个随机数，确定这个数是否在该范围中，这是通过用这个数除以列表中的每一个数并检查结果是否是1来实现的，用手选的这个值替代使用 **find()** 进行查找。
- 6-15 用在弧度制中表示角度的数填充 **vector<double>**。使用函数对象组成，产生 **vector** 中的所有元素的正弦（见 **<cmath>** 头文件）。
- 6-16 测试读者所使用的计算机的速度。调用 **srand(time(o))**，然后建一个随机数的数组。再次调用 **srand(time(o))**，并在第2个数组中生成相同个数的随机数。用 **equal()** 来

看两个数组是否相同。(如果你的计算机足够快的话,两次调用**time(o)**将返回相同的值。)如果两个数组内容不相同,对它们进行排序,并使用**mismatch()**来看看它们到底哪里不相同。如果相同,增加数组的长度并再次测试。

- 6-17 创建一个STL风格的算法**transform\_if()**,它遵循**transform()**的第1种形式,即仅在满足一元判定函数的对象上执行变换。将不满足判定函数的对象从结果中忽略掉。需要返回一个新的“末端”迭代器。
- 6-18 创建一个STL风格的**for\_each()**的重载形式算法,它遵循**transform()**的第2种形式。它用两个输入范围,这样就可以将第2个输入范围的对象传递给一个二元函数,对第1个范围中的每个对象应用这个函数。
- 6-19 创建一个由**vector<vector<T>>**制造的**Matrix**类模板。用提供给它的一个友元**ostream & operator<<(ostream&,const Matrix&)**来显示矩阵。在可能的地方用STL函数对象创建以下二元运算:**operator+(const Matrix&,const Matrix&)**执行矩阵加法,**operator\*(const Matrix&,const vector<int>&)**用一个**vector**乘一个矩阵,**operator\*(const Matrix&,const Matrix&)**执行矩阵乘法。(如果忘记了它们的运算规则,可能需要查找一下矩阵运算的数学含义。)使用**int**和**float**测试建立的**Matrix**类模板。
- 6-20 使用以下的字符

```
"~!@#%&^*()_+=}{|\\;":<.>?/"
```

生成一个密码本,以命令行给定的输入文件作为单词字典文件。不考虑排除非字母的字符,也不考虑单词在字典文件中的语境意义等情况。使每一种字符串的排列映射为一个单词,例如:

```
"=)/%[]|{*@?!";>&^~_:$+.#(<\ " apple
"|]\~>#.+(/[_`';={*"^!&?),@<" carrot
"@=~['].\/<-`>#*)^%+,";&?!_{:|$}(" Carrot
```

等等。

确认在密码本中不存在副本(相同)的密码或单词。使用**lexicographical\_compare()**在密码上执行排序。用密码本把字典文件译成密码。再对编码的字典文件进行解码,并确认得到的解码文件是否与原文件有相同的内容。

- 6-21 用下面的名字

```
Jon Brittle
Jane Brittle
Mike Brittle
Sharon Brittle
George Jensen
Evelyn Jensen
```

找到一个为这些人安排婚礼照片的所有可能的方法。

- 6-22 在区分照片后,每对新娘和新郎都希望所有其他照片上的人们作为来宾一起参加他们的婚礼。例如,如果新娘和新郎(Jon Brittle和Jane Brittle)相邻,找出为照片上的这对新人安排来宾的所有可能方法。
- 6-23 一家旅行社想要找出游客们从一个大陆的一端旅行到另一端(贯穿这个大陆)所花费的平均天数。问题是在调查中,一些游客不采用直接的路线,所用的时间往往要比需要的多(这样的例外数据点称为“局外点”)。使用下面的发生器,在一个**vector**上产生旅行

天数。使用**remove\_if()**删除**vector**中的所有的局外点。用**vector**中数据的平均值找出一般要花多长时间才能够完成旅行。

```
int travelTime() {
    // The "outlier"
    if(rand() % 10 == 0)
        return rand() % 100;
    // Regular route
    return rand() % 10 + 10;
}
```

- 6-24 在对一个已排序的序列范围进行查找时，确定采用**binary\_search()**（二分查找）要比**find()**（顺序查找）有多快。
- 6-25 某军队想在供选择的服役报名名单中征募新兵。他们已经决定征募那些在1997年报名注册应征的人们，按照出生日期，从年龄最大的开始依次征募直至最年轻的。在**vector**中产生任意数量的人（提供数据成员，如**age**和**yearEnrolled**）。划分**vector**，使那些在1997年登记注册的应征新兵在名单的开始位置，按照从最年轻的到年龄最大的顺序排序，名单中其余的部分按年龄从大到小排序。
- 6-26 用人口、（海拔）高度和天气等数据成员建一个名为**Town**的**class**。天气由一个**enum**用枚举常量表 **{RAINY,SNOWY,CLOUDY,CLEAR}** 建立。建一个产生**Town**对象的类。生成城镇的名称（采用读者自己起的有意义或者与地域无关的名称都行）或是从互联网上相关网站得来。保证全部的城镇名称是小写字母，并且没有重复。为了简便起见，我们建议保持城镇名称为一个词。为人口、海拔高度和天气字段，创建一个发生器，随机产生天气情况，在范围[100,1 000 000)内的人口及[0,8 000)英尺<sup>①</sup>内的海拔。用**Town**对象填充**vector**。把**vector**重新写入一个名为**Towns.txt**的新文件。
- 6-27 有一个生育高峰，导致了每个城镇人口按10%的速度增长。使用**transform()**更新这些城镇数据，并将数据重新写回文件中。
- 6-28 用最高和最低人口来查找这些城镇。这个练习对**Town**类实施**operator<**操作。并尝试实现一个函数，该函数当第1个参数小于第2个参数时返回**true**。将它作为所使用算法的判定函数。
- 6-29 找出所有海拔在2 500~3 500英尺间的城镇。根据需要对**Town**类执行相关运算。
- 6-30 现在需要在某个海拔高度的地方建一个飞机场，而场地位置不是问题。整理现有的城镇名单使其没有副本（副本意味着：在相同的100英尺范围中不能有两个海拔。例如这样的类包括 [100,199)、[200,199)等等）。使用**<functional>**中的函数对象，至少用两种不同的方式将名单按升序排列。以降序完成相同的工作。根据需要对**Town**实现相关运算。
- 6-31 在基于栈的数组中产生一组任意数目的随机数。使用**max\_element()**找到数组中最大的数。将它与数组末尾的数进行交换。找到次最大的数并且放在先前的数之前。持续这样做直到所有的元素都被移动过。当算法结束时，就得到一个排好序的数组。（这就是“选择排序”。）
- 6-32 编写一个程序，从一个文件中提取电话号码（同时也包括名字以及其他需要的信息），并且将以222开始的电话号码改变为以863开始。同时要保存旧的号码。文件形式如下所示：

① 1英尺=0.305m。

222 8945

756 3920

222 8432

等等。

- 6-33 编写一个程序，给定一个姓氏，找出每个有这个姓氏的人以及他或她的电话号码。使用处理序列范围的算法（例如**lower\_bound**、**upper\_bound**、**equal\_range**等等）。以姓氏作为主关键字，名字作为次关键字进行排序。假定从形式如下的文件中读入姓名及电话号码。（对它们进行排序，先按姓氏排好序，在同姓氏的人们中再按名字排好序。）

John Doe	345 9483
Nick Bonham	349 2930
Jane Doe	283 2819

- 6-34 给定一个包含类似下面数据的文件，将所有州的首字母省略词提取出来并将其放入一个单独的文件中。（注意，不能为某个数据类型决定该数据所占的行数，数据所占的行数是随机的。）

ALABAMA  
AL  
AK  
ALASKA  
ARIZONA  
AZ  
ARKANSAS  
AR  
CA  
CALIFORNIA  
CO  
COLORADO

等等。

当完成时，会得到一个含所有州的首字母省略词组成的文件，如下所示：

AL AK AZ AR CA CO CT DE FL GA HI ID IL IN IA KS KY LA ME MD  
MA MI MN MS MO MT NE NV NH NJ NM NY NC ND OH OK OR PA  
RI SC SD TN TX UT VT VA WA WV WI WY

- 6-35 创建一个**Employee**类，该类含有两个数据成员：**hours**和**hourlyPay**。**Employee**还含有一个返回雇员薪水的函数**calcSalary()**。对任意数量的雇员产生随机的小时薪水及工作小时数。用一个**vector<Employee\*>**来存放这些数据。查看一下公司将为这段付薪时期花多少钱。
- 6-36 再次相互比较**sort()**、**partial\_sort()**和**nth\_element()**函数，请查明如果都需要使用它们，那么使用其中的那一个进行弱排序可以更节省时间。

## 通用容器

容器类是一种特定代码重用问题的解决方案。它们是为了创建面向对象程序的构件，使程序内部模块的构建变得非常容易。

一个容器类描述了一个持有其他对象的对象。容器类如此重要以至于它们被认为是早期面向对象语言的基础。比如在Smalltalk中，程序员将编程语言看做一种与类库结合起来的翻译程序，而类库的关键就是容器类的集合。因此，C++编译器的供应商们很自然地也把容器类库包含在编译器中。读者将会注意到，本教材第1卷中以最简单的形式介绍过的**vector**是多么的有用。

就像很多其他早期的C++库一样，早期的容器类库遵循了Smalltalk的基于对象的层次结构(object-based hierarchy)，这种结构在Smalltalk中工作得很好，但是它在C++中却变得如此笨拙而难以使用。这就需要另外的解决方法。

C++中处理容器是采用基于模板的方式。标准C++库中的容器提供了多种数据结构。这些数据结构可以与标准算法一起很好地工作，来满足常见的软件开发需求。

### 7.1 容器和迭代器

在解决一个特定的问题时，如果不知道到底需要多少个对象，或这些对象将要维持多长时间，也就不能预先知道怎样存储这些对象。而在程序实际运行前你并不知道要创建多大的存储空间。

在面向对象程序设计中大多数这样的问题解决起来似乎很简单；只须创建对象的另一种类型就可以了。对于存储问题，这种新的对象类型持有其他对象或者是指向这些对象的指针。这种新的对象类型，通常在C++中称为容器（在一些语言中也称为收集器（collection），每当必须适应放置它内部的所有对象的需要的时候，容器都会自行扩展。所以不必预先知道容器中将要放入多少个对象；仅需要创建一个容器对象，然后由容器来处理全部细节。

幸运的是，一个好的面向对象编程语言都伴随着一个容器集。在C++中，它就是标准模板库（STL）。在某些库中，人们认为一个好的通用容器应该能够满足所有的需要，而在其他库中（特别是C++中）则针对不同的需要有不同的类型的容器：一个**vector**用于高效地访问其中的所有元素，而一个链表**list**则用于高效地在其中的所有位置上进行插入操作，还有更多其他类型的容器，所以人们可以根据自己的需要来选择特定类型的容器。

所有的容器都有某种存入对象和取出对象的方法。将某一对象放进一个容器的方法是十分明显的；可用一个名为“压入”或“增加”或者类似名字的函数。而从容器中检索对象的方法却并不总是明确的。如果这是一个类似数组的实体，比如一个**vector**，可以使用一个索引检索操作符或函数来完成。但是，在很多情况下这样做并没有意义。而且，单一选择函数也有其局限性。如果需要在容器中操纵或者比较一组元素时该怎么办呢？

对于灵活的元素访问的解决方案就是使用迭代器，迭代器是一个对象，它的工作就是在容器中挑选元素并将其呈献给迭代器的使用者。作为一个类，迭代器同时也提供了一个抽象层，因此可以将容器的内部实现细节与用来访问容器的代码分隔开来。通过迭代器，容器可以被看做一个序列。迭代器允许遍历一个序列而无需考虑基本结构——即不管它是一个**vector**、一个**list**、一个**set**还是其他结构。如此一来，就提供了这样的灵活性：即使在轻易地改变了底层的数据结构以后，也不会扰乱遍历容器的程序代码。将迭代操作从容器的控制下分隔开来，也



允许同时存在的多重迭代器。

从设计的观点来说，人们实际上想要做的事情不过就是需要一个序列，并能够操纵该序列以解决自己的问题。如果序列的一个类型可以满足所有要求的话，就没有必要使用不同的类型。基于两种原因需要在容器中进行选择。首先，各种容器提供了不同的接口类型和外部行为。**stack**具有与**queue**不同的接口和行为，对于一个**set**或一个**list**而言这也是不同的。其中的某一个容器可能比其他容器能够为问题提供更加灵活的解决方案，或者它能提供传达人们设计意图的更清晰的抽象。其次，不同的容器对于某些操作可能具有不同的效率。比如，在**vector**和**list**之间进行比较，效率就会有所不同。它们都是具有几乎相同的接口和外部行为的简单序列。但是某些操作却可能具有完全不同的代价。在**vector**中对元素的随机访问只是一个时间恒定的操作；无论选择哪一个元素它的时间代价都是一样的。然而，通过遍历的方式对一个**list**中的元素进行随机访问却是一个代价巨大的操作，元素在**list**中的位置越靠后，所需要的时间就越长。另一方面，如果要想向一个序列的中间插入一个元素，使用**list**的代价却比**vector**低。这些操作以及其他操作的效率依赖序列的底层结构。在设计阶段，可能开始时使用一个**list**，后来又在调整性能时转而使用**vector**，或者反过来。使用了迭代器，就使那些只遍历序列的代码与底层序列实现的改变隔离开来。

要记住的是，容器仅仅是一个存储对象的储存柜。如果那个储存柜满足了人们所有的要求，或许确实没有必要了解它是如何实现的。如果读者是在那种内在开销来自于其他因素的编程环境中工作的话，一个**vector**和一个**list**之间代价的差别也许就没那么重要了。可能的需要只是序列的一种类型。你甚至可以想象一种“完美”的容器抽象，它可以根据其使用方法自动地调整底层的实现。<sup>①</sup>

## STL参考文档

如前一章所述，读者也将注意到在本章中并没有包含用于描述每个STL容器的成员函数详尽的文档。虽然本章将描述我们使用到的成员函数，是我们没有给出其他成员函数的完整描述。我们推荐一些关于Dinkumware、Silicon Graphics以及STLPort STL实现的可利用的在线资源。<sup>②</sup>

## 7.2 概述

这里是一个使用**set**类模板的例子，一个模拟传统数学集合的容器，该容器不接受重复值。下面创建的**set**与**int**整型数据一起工作：

```
//: C07:Intset.cpp
// Simple use of STL set.
#include <cassert>
#include <set>
using namespace std;

int main() {
    set<int> intset;
    for(int i = 0; i < 25; i++)
        for(int j = 0; j < 10; j++)
            // Try to insert duplicates:
            intset.insert(j);
    assert(intset.size() == 10);
} ///:~
```

① 这是一个State模式的例子，将在第10章介绍。

② 请访问 <http://www.dinkumware.com>、<http://www.sgi.com/tech/stl>或 <http://www.stlport.org>。



成员函数**insert()**完成所有的工作：它试图插入一个元素并且如果容器中已经存在相同的元素则不予插入。在使用一个集合中涉及的操作通常只限于插入元素和检测集合是否包含要插入的元素。也可以形成一个并集、一个交集或者一个差集，并测试一个集合是否是另一个集合的子集。在这个例子中，值0~9被插入集合25次，但是只有10个惟一的实例被接受。

现在考虑使用**Intset.cpp**的形式并修改它，用以显示包含在一个文档中的单词清单。该解决方案变得非常简单。

```
//: C07:WordSet.cpp
#include <fstream>
#include <iostream>
#include <iterator>
#include <set>
#include <string>
#include "../require.h"
using namespace std;

void wordSet(const char* fileName) {
    ifstream source(fileName);
    assure(source, fileName);
    string word;
    set<string> words;
    while(source >> word)
        words.insert(word);
    copy(words.begin(), words.end(),
        ostream_iterator<string>(cout, "\n"));
    cout << "Number of unique words:"
        << words.size() << endl;
}

int main(int argc, char* argv[]) {
    if(argc > 1)
        wordSet(argv[1]);
    else
        wordSet("WordSet.cpp");
} ///:~
```

这里惟一的实质区别在于，集合保存字符串而不是整数。这些单词被从一个文件中取出来，但其他操作与**Intset.cpp**中的类似。该输出不仅显示出所有重复的单词都已经被忽略掉，而且由于**set**的实现方式，这些单词都被自动地排过序。

**set**是关联式容器（associative container）的一个例子，它是标准C++库提供的3种容器之一。下表列出了容器及其分类总结：

分 类	容 器
序列容器	<b>vector</b> 、 <b>list</b> 、 <b>deque</b>
容器适配器	<b>queue</b> 、 <b>stack</b> 、 <b>priority_queue</b>
关联式容器	<b>set</b> 、 <b>map</b> 、 <b>multiset</b> 、 <b>multimap</b>

这些分类表示，针对不同的需要使用不同的模型。序列容器仅将它们的元素线性地组织起来，是最基本的容器类型。对于某些问题，这些序列需要附上某些特殊的属性，这正好是容器适配器要做的事情——它们对诸如队列或者栈的抽象建立模型。关联式容器则基于关键字来组织它们的数据，并允许快速地检索那些数据。

标准库中所有的容器都持有存入的对象的拷贝，并且根据需要扩展它们的资源，所以这些对象都必须是可构造拷贝（copy-constructible）（具有一个可访问的拷贝构造函数）和可赋值（assignable）拷贝（具有一个可访问的赋值操作符）的。一个容器与其他容器之间的关键不同

之处在于它们在内存中存储对象的方式和向用户提供什么样的操作。

如读者已经知道的那样，**vector**是一种允许快速随机访问其中元素的线性序列。然而，向类似于**vector**这样排在一起的序列的中间插入一个元素的操作的开销却是很大的，就像对一个数组进行这种操作一样。一个**deque**（双端队列（double-ended-queue），读作“deck”）也允许几乎与**vector**一样快的随机访问，但是当需要分配新的存储空间时速度明显更快，而且很容易在序列的前端和后端加进新的元素。**list**是一个双向链表，所以在其上随机地移动某个元素的代价很高，但却可以用很低的代价向其中任何地方插入元素。因此，**list**、**deque**和**vector**在基本功能上很相似（它们都是线性序列），只是在各种操作的代价上有所不同。在一个程序的开始阶段可以选择它们中的任何一种使用，只在为了调整效率的时候尝试更换为其他容器。

很多问题的解决其实只需要一个像**list**、**deque**或**vector**这样简单的线性序列。所有这3个容器都含有用于向序列尾部插入一个元素的成员函数**push\_back()**（**list**和**deque**还有一个**push\_front()**成员，用于将一个元素插入序列前端）。

但是，如何在一个序列容器中检索存储的元素呢？对于**vector**和**deque**可以使用索引检索操作符**operator[]**，但对于**list**这是行不通的。这3种容器都可以使用迭代器来访问元素。每种容器都提供了相应类型的迭代器来访问它的元素。

虽然容器由值来保存对象（也就是说，它们持有对象的全部拷贝），而在某些时候希望容器存储一些指针，这些指针可以指向某一层结构对象，这样一来就可以利用类表现出的多态行为。考虑经典的“图形（shape）”例子，在这里所有的图形都有一个共同的操作集，而且拥有不同类型的图形。这里有一个程序例子，它看起来像使用STL **vector**来持有指向在堆中创建的不同类型**Shape**对象的指针：

```
//: C07:Stlshape.cpp
// Simple shapes using the STL.
#include <vector>
#include <iostream>
using namespace std;

class Shape {
public:
    virtual void draw() = 0;
    virtual ~Shape() {};
};

class Circle : public Shape {
public:
    void draw() { cout << "Circle::draw" << endl; }
    ~Circle() { cout << "~Circle" << endl; }
};

class Triangle : public Shape {
public:
    void draw() { cout << "Triangle::draw" << endl; }
    ~Triangle() { cout << "~Triangle" << endl; }
};

class Square : public Shape {
public:
    void draw() { cout << "Square::draw" << endl; }
    ~Square() { cout << "~Square" << endl; }
};

int main() {
    typedef std::vector<Shape*> Container;
```



```

typedef Container::iterator Iter;
Container shapes;
shapes.push_back(new Circle);
shapes.push_back(new Square);
shapes.push_back(new Triangle);
for(Iter i = shapes.begin(); i != shapes.end(); i++)
    (*i)->draw();
// ... Sometime later:
for(Iter j = shapes.begin(); j != shapes.end(); j++)
    delete *j;
} ///:~

```

类**Shape**、**Circle**、**Square**和**Triangle**的创建非常相似。**Shape**是一个抽象基类（因为有纯虚函数指明标记 `=0`），它定义了所有**Shape**类型的接口。派生类覆盖虚函数**virtual draw()**以实现相应的操作。现在要创建一串不同类型的**Shape**对象，并将它们原封不动存储在一个STL容器内。为方便起见，用类型定义：

```
typedef std::vector<Shape*> Container;
```

为**Shape\***的**vector**创建一个别名，而用类型定义：

```
typedef Container::iterator Iter;
```

使用前面定义的别名为**vector<Shape\*>::iterator**创建另一个别名。注意，容器类型名必须用于产生合适的迭代器，它被定义为一个嵌套类。虽然存在不同类型的迭代器（前向、双向、随机等等），但它们都拥有同一个基本接口：可以使用++对它们进行增1操作，可以对迭代器解析以便产生它们当前选中的对象，而且可以测试它们以查看是否已经到了序列的末尾。这就是在90%的时间里要做的事情。这也正是前面例子中所做的事情：一个容器被创建以后，它被填入不同类型的**Shape**指针。注意，向上类型转换发生在当**Circle**、**Square**或者**Rectangle**指针被加入**Shapes**容器中去的时候，容器并不知道加入的指针的具体类型，作为替代它只持有**Shape\***。一旦指针被装入容器，它就失去了明确的特性而成为了一个匿名的**Shape\***。这正是我们想要的：将它们全都投掷进容器，然后再利用多态性把它们挑选出来。

第1个**for**循环创建一个迭代器，并且通过调用容器的**begin()**成员函数将其设置为指向序列的开始端。所有的容器都有**begin()**和**end()**成员函数，分别用来产生选择序列开始端和超越末尾的迭代器。可以通过确认迭代器不等于通过调用**end()**函数产生的迭代器的办法来测试操作是否已经完成；不要使用 `<` 或者 `<=`。只有 `!=` 和 `==` 测试方式起作用，所以通常将循环写成如下形式：

```
for(Iter i = shapes.begin(); i != shapes.end(); i++)
```

这条语句的意思是“遍历序列中的每一个元素。”

对迭代器做什么才可以产生它所选择的元素呢？可以通过 `*`（这实际上是一个重载了的运算符）解析其引用（请读者思考其他方法）来实现。返回的是容器持有的任何东西。这个容器持有**Shape\***，所以这就是**\*i**所产生的结果。如果希望调用**Shape**的成员函数，必须使用操作符 `->`，因此写出下面的一行：

```
(*i)->draw();
```

这将会调用迭代器当前选择的**Shape\***的**draw()**成员函数。这里的括号虽然难看，但却产生运算符优先级所必需的。

当这些对象已经被销毁或者在其他情况下这些指针被删除时，STL容器并不会自动地为它们包含的指针调用**delete**。如果用**new**在堆中创建了一个对象，并将其指针存放到某个容器中，这个容器不会提示该指针是否同时也存入到了另一个容器，也不会提示它是否指向堆内存

中的开始位置。用户必须始终负责管理自己的堆内存的分配。程序中的最后一行遍历并且删除容器中所有的对象，以便彻底地实施清理工作。处理容器中指针最容易和最安全的办法就是使用智能（smart）指针。要注意的是**auto\_ptr**并不能用于这种目的，所以必须在C++标准库以外寻找适当的智能指针。<sup>①</sup>

可以修改这个程序中的两行以便改变本例中使用的容器类型。用包含**<list>**来替代包含**<vector>**，并且将第1个**typedef**改写如下：

```
typedef std::list<Shape*> Container;
```

以替代正在使用**vector**。对其他任何地方不做修改。可以这样做不是因为由继承强加了一个接口（在STL只有很少一点继承），而是因为按STL的设计者采用的惯例已经强加了接口，所以可以非常准确地进行这类交换。现在就可以很容易地在**vector**与**list**或是任何其他支持相同接口（语法和语义上均相同）的容器之间进行变换，并且看看对于需求来说哪种容器工作起来最快。

### 7.2.1 字符串容器

在前面的例子中，在**main()**的最后需要遍历整个的链表，并且用**delete**删除所有的**Shape**指针：

```
for(Iter j = shapes.begin(); j != shapes.end(); j++)
    delete *j;
```

STL容器确保在其自身被销毁时将调用其包含的每个对象的析构函数。然而，指针并没有析构函数，因此用户必须自己用**delete**删除它们。

这里明显看到STL中的一个疏漏：在任何STL容器中都没有自动用**delete**删除它们包含的指针的设施，所以必须人工地自行解决。这表明STL的设计者们似乎认为指针的容器并不是一个有趣的问题，但事实并不是这样的。

由于存在多重成员资格（multiple membership）问题，使得自动删除一个指针成为问题。如果一个容器持有一个指向某个对象的指针，并不表明那个指针就不会在另一个容器中出现。**Trash**指针链表中的一个指向**Aluminum**对象的指针，也可能存在于一个**Aluminum**指针链表中。如果发生了这种情况，哪个链表负责清理这个对象——即哪个链表“拥有”这个对象呢？

这个问题事实上可以通过在链表中存储对象而不是指针来解决。当链表被销毁的时候似乎它包含的对象也必须被销毁。在这里，当看到创建一个包含**string**型对象的容器时，STL表现出了它的闪光点。下面的例子将每一输入行作为一个**string**型字符串对象存入一个**vector<string>**中：

```
//: C07:StringVector.cpp
// A vector of strings.
#include <fstream>
#include <iostream>
#include <iterator>
#include <sstream>
#include <string>
#include <vector>
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
    const char* fname = "StringVector.cpp";
```

① 随着更多的smart指针类型将加入下一个版本的标准，情况将会发生变化。如果想要先了解它们，可以在[www.boost.org](http://www.boost.org)看到这些智能指针。

```

    if(argc > 1) fname = argv[1];
    ifstream in(fname);
    assure(in, fname);
    vector<string> strings;
    string line;
    while(getline(in, line))
        strings.push_back(line);
    // Do something to the strings...
    int i = 1;
    vector<string>::iterator w;
    for(w = strings.begin(); w != strings.end(); w++) {
        ostringstream ss;
        ss << i++;
        *w = ss.str() + ": " + *w;
    }
    // Now send them out:
    copy(strings.begin(), strings.end(),
        ostream_iterator<string>(cout, "\n"));
    // Since they aren't pointers, string
    // objects clean themselves up!
} ///:~

```

一旦名为**strings**的**vector<string>**被创建，文件中的每一行都作为一个**string**对象被读入并且放入**vector**中：

```

while(getline(in, line))
    strings.push_back(line);

```

对该文件的操作是向其中添加行号。一个**stringstream**对象提供了简便的方法将一个**int**型数字转换为一个用字符表示那个整数的**string**。

因为操作符**operator+**已被重载，所以组合**string**对象是相当容易的。非常合乎情理，迭代器**w**可以解析以便产生一个既可作为右值又可作为左值的字符串。

```
*w = ss.str() + ": " + *w;
```

通过迭代器可以反向给容器中的元素进行赋值，这可能会让人感到惊讶，但这就是对STL的精心设计做出的贡献。

因为**vector<string>**包含对象，这里有两件事值得注意。第一，就像前面解释过的那样，不必明确地清理**string**对象。即便将指向这些**string**对象的地址作为指针放入其他容器也一样，显而易见**strings**就是“主链表”，它拥有对那些对象的所有权。

第二，有效地使用对象的动态创建方法，并且还绝不使用**new**或者**delete**！因为已经保存了给予它的对象拷贝，所有这些问题交由**vector**进行处理。因此能够有效地清理编码。

## 7.2.2 从STL容器继承

那种即刻就能创建一个元素序列的能力是令人惊异的，这使人们回想起以前为解决这个特殊的问题时花费了多少时间。比如，很多实用的程序都包括这样的功能，将一个文件读进内存，修改该文件，然后再写回到磁盘上。读者也可以用**StringVector.cpp**中的那些功能并将其打包成一个类，以便以后使用。

现在的问题是：在程序设计中，是创建一个**vector**类型的成员对象，还是采用继承的方式派生出一个类似的对象？一般情况下，面向对象设计准则更倾向于使用组合（成员对象）而不是继承，但是有些标准算法盼望有这样一些序列，它们实现某个特殊的接口，因此继承的使用常常是必需的。

```

//: C07:FileEditor.h
// A file editor tool.
#ifndef FILEEDITOR_H
#define FILEEDITOR_H
#include <iostream>
#include <string>
#include <vector>

class FileEditor : public std::vector<std::string> {
public:
    void open(const char* filename);
    FileEditor(const char* filename) { open(filename); }
    FileEditor() {};
    void write(std::ostream& out = std::cout);
};
#endif // FILEEDITOR_H ///:~

```

构造函数打开文件，将其读入到**FileEditor**，再用成员函数**write()**将包含**string**的**vector**写入任何一个输出流**ostream**。注意，在**write()**中可以使用一个默认的参数作为引用。

其实现相当简单：

```

//: C07:FileEditor.cpp {0}
#include "FileEditor.h"
#include <fstream>
#include "../require.h"
using namespace std;

void FileEditor::open(const char* filename) {
    ifstream in(filename);
    assure(in, filename);
    string line;
    while(getline(in, line))
        push_back(line);
}

// Could also use copy() here:
void FileEditor::write(ostream& out) {
    for(iterator w = begin(); w != end(); w++)
        out << *w << endl;
} ///:~

```

来自**StringVector.cpp**的函数在这里仅被重新打包。这是类进化的常用方法——程序员在开始时创建一个程序来解决某一特殊的应用，然后发现其中有些通用的功能，就可以把它们变成一个类。

现在，那个行号产生程序可以用**FileEditor**重新编写如下：

```

//: C07:FEditTest.cpp
//{L} FileEditor
// Test the FileEditor tool.
#include <sstream>
#include "FileEditor.h"
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
    FileEditor file;
    if(argc > 1) {
        file.open(argv[1]);
    } else {
        file.open("FEditTest.cpp");
    }
    // Do something to the lines...
}

```



```

int i = 1;
FileEditor::iterator w = file.begin();
while(w != file.end()) {
    ostringstream ss;
    ss << i++;
    *w = ss.str() + ": " + *w;
    ++w;
}
// Now send them to cout:
file.write();
} ///:~

```

现在，用于读取文件的操作都在构造函数中了：

```
FileEditor file(argv[1]);
```

(或者在成员函数**open()**中)，而写操作发生在单独的一行中（默认将数据发送输出到**cout**）：

```
file.write();
```

该程序块被包含进来用以对内存中的文件进行修改。

### 7.3 更多迭代器

迭代器是为实现通用而做的抽象。它与不同类型的容器一起工作而不必了解那些容器的底层结构。绝大多数容器都支持迭代器，<sup>①</sup>所以可以像下面这样：

```

<ContainerType>::iterator
<ContainerType>::const_iterator

```

为一个容器创建迭代器类型。每一个容器都有一个起始成员函数**begin()**以产生指向容器中起始元素的迭代器，和一个末尾成员函数**end()**用以产生容器的超越末尾的标记迭代器。如果容器是一个**const**（常）容器，则**begin()**和**end()**产生**const**（常）迭代器，即不允许更换这些迭代器所指向的元素（因为相应的运算符都是**const**的）。

所有的迭代器都可以在它们的序列中向前移动（通过运算符**operator++**），并且允许使用**==**和**!=**进行比较。因此，为在不超出范围的前提下前移一个名为**it**的迭代器，可以进行如下处理：

```

while(it != pastEnd) {
    // Do something
    ++it;
}

```

这里**pastEnd**是由容器的成员函数**end()**产生的超越末尾的标记。

通过使用解析运算符（**operator\***），一个迭代器可用于产生其当前所指的容器元素。这可以有两种形式。如果**it**是一个可以遍历容器的迭代器，并且**f()**是容器持有的对象类型的一个成员函数，就可以使用两种形式中的任一种形式：

```
(*it).f();
```

或者

```
it->f();
```

了解了这些以后，就可以创建一个可以与任何容器一起工作的模板了。在这里，函数模板**apply()**为容器中的每个对象调用一个成员函数，它使用一个指向成员函数的指针作为参数进行传递：

<sup>①</sup> 容器适配器、栈、队列和优先队列不支持迭代器，因为在用户看来它们的行为与序列的行为并不相同。

```

//: C07:Apply.cpp
// Using simple iteration.
#include <iostream>
#include <vector>
#include <iterator>
using namespace std;

template<class Cont, class PtrMemFun>
void apply(Cont& c, PtrMemFun f) {
    typename Cont::iterator it = c.begin();
    while(it != c.end()) {
        ((*it).*f)(); // Alternate form
        ++it;
    }
}

class Z {
    int i;
public:
    Z(int ii) : i(ii) {}
    void g() { ++i; }
    friend ostream& operator<<(ostream& os, const Z& z) {
        return os << z.i;
    }
};

int main() {
    ostream_iterator<Z> out(cout, " ");
    vector<Z> vz;
    for(int i = 0; i < 10; i++)
        vz.push_back(Z(i));
    copy(vz.begin(), vz.end(), out);
    cout << endl;
    apply(vz, &Z::g);
    copy(vz.begin(), vz.end(), out);
} ///:~

```

在这里不能使用**operator->**，因为这将导致语句成为：

```
(it->*f)();
```

它将尝试使用迭代器的**operator->\***，而该操作符在迭代器类中并未提供。<sup>①</sup>

就像在第6章中所看到的那样，可以更容易地使用**for\_each()**或者**transform()**两者之中任一个函数应用到序列。

### 7.3.1 可逆容器中的迭代器

一个容器也可以是可逆的 (reversible)，这意味着容器可以产生一个从末尾反向移动的迭代器，这些迭代器也可以从容器的起始元素前向移动。所有标准的容器都支持这种双向迭代。

可逆容器拥有成员函数**rbegin()**（用于产生一个选择了容器末尾的迭代器**reverse\_iterator**）和**rend()**（用于产生一个指向“超越起始”的迭代器**reverse\_iterator**）。如果容器为**const**容器，则**rbegin()**和**rend()**将会产生**const\_reverse\_iterator**。

下面的例子使用**vector**，但该例适用于所有支持迭代操作的容器：

① 它仅仅适用于那些使用了一个 (**a T\***) 指针作为迭代器类型的**vector**的实现，就像STLPort所做的那样。



```

//: C07:Reversible.cpp
// Using reversible containers.
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
#include "../require.h"
using namespace std;

int main() {
    ifstream in("Reversible.cpp");
    assure(in, "Reversible.cpp");
    string line;
    vector<string> lines;
    while(getline(in, line))
        lines.push_back(line);
    for(vector<string>::reverse_iterator r = lines.rbegin();
        r != lines.rend(); r++)
        cout << *r << endl;
} ///:~

```

反向移动遍历一个容器，使用与一个前向移动遍历容器的普通的迭代器时相同的语法。

### 7.3.2 迭代器的种类

标准C++库中的迭代器被归类为若干“种类”以便描述它们的性能。通常对它们的描述顺序是从行为约束最严格的种类到行为功能最强大的种类。

#### 1. 输入迭代器：只读，一次传递

为输入迭代器的预定义实现只有**istream\_iterator**和**istreambuf\_iterator**，用于从一个输入流**istream**中读取。就像想象的那样，一个输入迭代器仅能对它所选择的每个元素进行一次解析，正如只能对一个输入流的特殊部分读取一次一样。它们只能前向移动。一个专门的构造函数定义了超越末尾的值。总之，输入迭代器可以对读操作的结果进行解析（对每一个值仅解析一次），然后前向移动。

#### 2. 输出迭代器：只写，一次传递

这是对输入迭代器的补充，不过是对写操作而不是读操作。为输出迭代器的预定义实现只有**ostream\_iterator**和**ostreambuf\_iterator**，用于向一个输出流**ostream**写数据，还有一个一般较少使用的**raw\_storage\_iterator**。再次强调，它们只能对每个写出的值进行一次解析，并且只能前向移动。对于输出迭代器来说，没有使用超越末尾的值来结束的概念。总之，输出迭代器可以对写操作的值进行解析（对每一个值仅解析一次），然后前向移动。

#### 3. 前向迭代器：多次读/写

前向迭代器包含了输入和输出迭代器两者的所有功能，加上还可以多次解析一个迭代器指定的位置，因此可以对一个值进行多次读/写。顾名思义，前向迭代器只能向前移动。没有专为前向迭代器预定义的迭代器。

#### 4. 双向迭代器：operator--

双向迭代器具有前向迭代器的全部功能，另外它还可以利用自减操作符**operator--**向后一次移动一个位置。由**list**容器中返回的迭代器都是双向的。

#### 5. 随机访问迭代器：类似于一个指针

最后，随机访问迭代器具有双向迭代器的所有功能，再加上一个指针所有的功能（一个指针就是一个随机访问迭代器），除了没有一种“空（null）”迭代器和空指针相对应。基本上可以这样说，一个随机访问迭代器就像一个指针那样可以进行任何操作，包括使用操作符

**operator[]**进行索引，加某个整数值到一个指针就可以向前或向后移动若干个位置，或者使用比较运算符在迭代器之间进行比较。

#### 6. 重要性

人们为什么要关心这些分类法？当只需要以一种明确的方式（比如，仅仅是手工编码想要对某个容器中的对象进行所有的操作）来使用容器时，这些分类法通常并不重要。那么，对迭代器进行分类到底有什么用处。迭代器种类在下列场合中就很重要：

1) 使用某些不久就要演示的C++爱好者内置的迭代器类型，或者用户已经“学成毕业”，能够胜任创建自己的迭代器的工作（在本章稍后演示）。

2) 使用STL算法（第6章的主题）。每种算法对其迭代器都有使用场合的要求。在创建用户自己的可重用的算法模板的时候，迭代器种类的知识变得尤其重要，因为自定义算法需要的迭代器种类决定了该算法的灵活性。如果仅要求最基本的迭代器种类（输入或者输出迭代器），这种算法则适合于任何场合（**copy()**就是这样的例子）。

一个迭代器的种类由一个迭代器的层次结构标记类进行标识。类名和迭代器的种类相符合，并且它们之间的派生层次结构反映了它们之间的关系：

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag :
    public input_iterator_tag {};
struct bidirectional_iterator_tag :
    public forward_iterator_tag {};
struct random_access_iterator_tag :
    public bidirectional_iterator_tag {};
```

类**forward\_iterator\_tag**仅从**input\_iterator\_tag**派生，而不是从**output\_iterator\_tag**派生，因为在使用前向迭代器的算法中需要超越末尾的迭代器值，但是使用输出迭代器的那些算法总是假定运算符**operator\***是可以解析的。为了这个原因，保证一个超越末尾的值绝不会传递到那些希望使用输出迭代器的算法中是很重要的。

为提高效率，某些算法为不同种类的迭代器提供不同的实现，这些迭代器是从由其定义的迭代器标记中推算出来的。在本章稍后部分当我们创建自己的迭代器类型时，将会用到这些标记类。

### 7.3.3 预定义迭代器

STL拥有一个用起来相当便利的预定义迭代器的集合。正如所见到的，对所有基本容器调用**rbegin()**和**rend()**可得到**reverse\_iterator**对象。

因为某些STL算法需要插入迭代器（insertion iterator）——例如，**copy()**算法——使用赋值操作符**operator=**将对象放进目的容器中去。在向容器中填充（fill）而不是覆盖已经存在于目的容器中的那些元素时，当在那里已经没有空间可填充的时候，就会产生问题。插入迭代器所做的事情就是改变运算符**operator=**的实现来替代赋值操作，称为该容器的“压入”或“插入”函数，因此该函数就引起容器分配新的存储空间。**back\_insert\_iterator**和**front\_insert\_iterator**两者的构造函数都使用一个基本序列容器对象（**vector**、**deque**或**list**）作为其参数，并产生一个分别调用**push\_back()**或**push\_front()**以进行赋值的迭代器。有益的函数**back\_inserter()**和**front\_inserter()**让程序员在产生这些插入迭代器对象的时候少写一些代码。因为所有的基本序列容器都支持**push\_back()**，读者可能会发现，使用**back\_inserter()**已经成为某种经常性的工作。

**insert\_iterator**能够向一个序列的中间插入元素，再一次代替了**operator=**的含义，

但是这一次则是自动调用插入函数**insert()**而不是某个“压入”函数。**insert()**成员函数需要一个迭代器在插入前指定插入的位置，所以除了容器对象以外，**insert\_iterator**还需要这个迭代器。插入器函数**inserter()**产生相同的对象。

下面的例子演示了不同类型的插入器的使用：

```

//: C07:Inserters.cpp
// Different types of iterator inserters.
#include <iostream>
#include <vector>
#include <deque>
#include <list>
#include <iterator>
using namespace std;

int a[] = { 1, 3, 5, 7, 11, 13, 17, 19, 23 };

template<class Cont> void frontInsertion(Cont& ci) {
    copy(a, a + sizeof(a)/sizeof(Cont::value_type),
        front_inserter(ci));
    copy(ci.begin(), ci.end(),
        ostream_iterator<typename Cont::value_type>(
            cout, " "));
    cout << endl;
}

template<class Cont> void backInsertion(Cont& ci) {
    copy(a, a + sizeof(a)/sizeof(Cont::value_type),
        back_inserter(ci));
    copy(ci.begin(), ci.end(),
        ostream_iterator<typename Cont::value_type>(
            cout, " "));
    cout << endl;
}

template<class Cont> void midInsertion(Cont& ci) {
    typename Cont::iterator it = ci.begin();
    ++it; ++it; ++it;
    copy(a, a + sizeof(a)/(sizeof(Cont::value_type) * 2),
        inserter(ci, it));
    copy(ci.begin(), ci.end(),
        ostream_iterator<typename Cont::value_type>(
            cout, " "));
    cout << endl;
}

int main() {
    deque<int> di;
    list<int> li;
    vector<int> vi;
    // Can't use a front_inserter() with vector
    frontInsertion(di);
    frontInsertion(li);
    di.clear();
    li.clear();
    backInsertion(vi);
    backInsertion(di);
    backInsertion(li);
    midInsertion(vi);
    midInsertion(di);
    midInsertion(li);
} ///:~

```



因为**vector**不支持**push\_front()**，所以它不能产生一个**front\_insert\_iterator**。然而，可以看到**vector**支持另外两种插入的类型（即便如此，稍后也将看到对于**vector**来说**insert()**并不是一个高效的操作）。注意，这里用嵌套类型**Cont::value\_type**而不是硬编码的**int**类型。

### 1. 更多的流迭代器

在第6章中，结合**copy()**函数介绍了流迭代器**ostream\_iterator**（输出迭代器）和**istream\_iterator**（输入迭代器）的用法。要记住，输出流是没有“结束”这个概念的，因为用户总是在持续地写出更多的元素。然而，输入流却最终会结束（比如，达到了文件末尾），所以需要有一种方法来表现这一点。**istream\_iterator**有两个构造函数，一个获得输入流**istream**并且产生一个实际读取的迭代器，另一个是默认构造函数，用于产生一个作为超越末尾的标记的对象。在下面的例子中这个对象被命名为**end**：

```
//: C07:StreamIt.cpp
// Iterators for istreams and ostream.
#include <fstream>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>
#include "../require.h"
using namespace std;

int main() {
    ifstream in("StreamIt.cpp");
    assure(in, "StreamIt.cpp");
    istream_iterator<string> begin(in), end;
    ostream_iterator<string> out(cout, "\n");
    vector<string> vs;
    copy(begin, end, back_inserter(vs));
    copy(vs.begin(), vs.end(), out);
    *out++ = vs[0];
    *out++ = "That's all, folks!";
} ///:~
```

当**in**用完输入时（在这个例子中，是指到达了文件的末尾），**init**与**end**相等，于是**copy()**终止。

因为**out**是一个**ostream\_iterator<string>**，使用运算符**operator=**可以分配任何**string**对象给解析后的迭代器，并且将那个**string**放入输出流中，就像在两个给**out**赋值的操作所做的那样。因为**out**在定义时以一个新行作为其第2个参数，所以这个赋值操作也在每次赋值时插入一个新行。

虽然可能创建一个**istream\_iterator<char>**和**ostream\_iterator<char>**，但实际上这样做会从语法上分析（parse）输入并且导致诸如自动地吃掉空白字符（空格、制表符和换行符），如果希望用一个输入流的精确地表现这样的动作，是不可取的。另一种方法可以使用特殊的迭代器**istreambuf\_iterator**和**ostreambuf\_iterator**，它们被设计用来严格地移动字符。<sup>①</sup>虽然这些都是模板，但它们都想要使用**char**或者**wchar\_t**作为模板参数。<sup>②</sup>在下面的例子中，让我们来比较流迭代器和流缓冲迭代器的行为：

① 创建这些迭代器实际上是为了将现场面从输入输出流中抽象出来，从而使现场面能够处理任何字符序列，不仅仅是输入输出流。这样现场允许输入输出流可以轻易地处理一些不同的格式（比如货币符号的表示方式）。

② 对于其他参数类型，用户需要提供一个用于特化的**char\_traits**。

```

//: C07:StreambufIterator.cpp
// istreambuf_iterator & ostreambuf_iterator.
#include <algorithm>
#include <fstream>
#include <iostream>
#include <iterator>
#include "../require.h"
using namespace std;

int main() {
    ifstream in("StreambufIterator.cpp");
    assure(in, "StreambufIterator.cpp");
    // Exact representation of stream:
    istreambuf_iterator<char> isb(in), end;
    ostreambuf_iterator<char> osb(cout);
    while(isb != end)
        *osb++ = *isb++; // Copy 'in' to cout
    cout << endl;
    ifstream in2("StreambufIterator.cpp");
    // Strips white space:
    istream_iterator<char> is(in2), end2;
    ostream_iterator<char> os(cout);
    while(is != end2)
        *os++ = *is++;
    cout << endl;
} ///:~

```

从语法上来分析，流迭代器使用由**`istream::operator>`**来定义，如果读者正在直接从语法上分析字符的话这也许不是你所希望的——要从字符流中将所有的空白字符都去掉的做法相当罕见。事实上读者总希望在使用字符和流的时候使用流缓冲迭代器，而不是使用流迭代器。另外，**`istream::operator>`**为每次操作增加了不小的开销，所以它只适合于较高级的操作，比如从语法上分析数字。<sup>①</sup>

## 2. 操纵未初始化的存储区

**`raw_storage_iterator`**在**`<memory>`**中定义，它是一个输出迭代器。它提供了使算法能够将其结果存储到未经初始化的内存的能力。其接口相当简单：构造函数持有一个指向某原始（未初始化）内存存储区的迭代器（典型的指针），并且运算符**`operator=`**将一个对象分配给那个原始内存。模板参数是输出迭代器的类型和将要被存储的对象类型，输出迭代器指向该原始存储区。这里的例子创建了**Noisy**对象，它们打印出这些对象的构造、赋值以及析构时的跟踪语句（将在稍后介绍**Noisy**类的定义）：

```

//: C07:RawStorageIterator.cpp {-bor}
// Demonstrate the raw_storage_iterator.
//{L} Noisy
#include <iostream>
#include <iterator>
#include <algorithm>
#include "Noisy.h"
using namespace std;

int main() {
    const int QUANTITY = 10;
    // Create raw storage and cast to desired type:
    Noisy* np = reinterpret_cast<Noisy*>(
        new char[QUANTITY * sizeof(Noisy)]);

```

① 我们应该感谢Nathan Myers对此的解释。



```

raw_storage_iterator<Noisy*, Noisy> rsi(np);
for(int i = 0; i < QUANTITY; i++)
    *rsi++ = Noisy(); // Place objects in storage
cout << endl;
copy(np, np + QUANTITY,
    ostream_iterator<Noisy>(cout, " "));
cout << endl;
// Explicit destructor call for cleanup:
for(int j = 0; j < QUANTITY; j++)
    (&np[j])->~Noisy();
// Release raw storage:
delete reinterpret_cast<char*>(np);
} ///:~

```

为了能够正确地使用**raw\_storage\_iterator**模板，原始存储区类型必须与所创建的对象类型相同。这就是为什么来自新**char**数组的指针被类型转换为**Noisy\***的原因。赋值操作符使用拷贝构造函数将对象强制存入原始存储区。注意，必须显式地调用析构函数以便进行适当的清理工作，这也允许在操纵容器期间每次删除一个对象。但表达式**delete np**无论如何是无效的，因为在**delete**表达式中的一个静态指针类型，必须与**new**表达式中分配的类型相同。

## 7.4 基本序列容器：vector、list和deque

所有基本序列容器完全按照存进去时的顺序持有对象。然而，对于不同的基本序列容器，它们的操作效率是不同的，因此如果想要操纵具有某种特点的序列，则应当针对不同的操作类型选择合适的容器。到现在为止，本教材中已经使用了**vector**作为精选的容器。并经常在一些示例中应用它。然而，当开始用容器做更复杂的工作时，更多地了解容器的底层实现和行为就变得很重要了，这样就使得程序员能够根据需要做出正确的选择。

### 7.4.1 基本序列容器的操作

下面的例子用一个模板演示了所有基本序列容器：**vector**、**deque**和**list**所支持的操作：

```

//: C07:BasicSequenceOperations.cpp
// The operations available for all the
// basic sequence Containers.
#include <deque>
#include <iostream>
#include <list>
#include <vector>
using namespace std;

template<typename Container>
void print(Container& c, char* title = "") {
    cout << title << ':' << endl;
    if(c.empty()) {
        cout << "(empty)" << endl;
        return;
    }
    typename Container::iterator it;
    for(it = c.begin(); it != c.end(); it++)
        cout << *it << " ";
    cout << endl;
    cout << "size() " << c.size()
        << " max_size() " << c.max_size()
        << " front() " << c.front()
        << " back() " << c.back()
        << endl;
}

```



```

template<typename ContainerOfInt> void basicOps(char* s) {
    cout << "----- " << s << " -----" << endl;
    typedef ContainerOfInt Ci;
    Ci c;
    print(c, "c after default constructor");
    Ci c2(10, 1); // 10 elements, values all 1
    print(c2, "c2 after constructor(10,1)");
    int ia[] = { 1, 3, 5, 7, 9 };
    const int IASZ = sizeof(ia)/sizeof(*ia);
    // Initialize with begin & end iterators:
    Ci c3(ia, ia + IASZ);
    print(c3, "c3 after constructor(iter,iter)");
    Ci c4(c2); // Copy-constructor
    print(c4, "c4 after copy-constructor(c2)");
    c = c2; // Assignment operator
    print(c, "c after operator=c2");
    c.assign(10, 2); // 10 elements, values all 2
    print(c, "c after assign(10, 2)");
    // Assign with begin & end iterators:
    c.assign(ia, ia + IASZ);
    print(c, "c after assign(iter, iter)");
    cout << "c using reverse iterators:" << endl;
    typename Ci::reverse_iterator rit = c.rbegin();
    while(rit != c.rend())
        cout << *rit++ << " ";
    cout << endl;
    c.resize(4);
    print(c, "c after resize(4)");
    c.push_back(47);
    print(c, "c after push_back(47)");
    c.pop_back();
    print(c, "c after pop_back()");
    typename Ci::iterator it = c.begin();
    ++it; ++it;
    c.insert(it, 74);
    print(c, "c after insert(it, 74)");
    it = c.begin();
    ++it;
    c.insert(it, 3, 96);
    print(c, "c after insert(it, 3, 96)");
    it = c.begin();
    ++it;
    c.insert(it, c3.begin(), c3.end());
    print(c, "c after insert("
        "it, c3.begin(), c3.end())");
    it = c.begin();
    ++it;
    c.erase(it);
    print(c, "c after erase(it)");
    typename Ci::iterator it2 = it = c.begin();
    ++it;
    ++it2; ++it2; ++it2; ++it2; ++it2;
    c.erase(it, it2);
    print(c, "c after erase(it, it2)");
    c.swap(c2);
    print(c, "c after swap(c2)");
    c.clear();
    print(c, "c after clear()");
}

int main() {
    basicOps<vector<int> >("vector");
}

```



```

    basicOps<deque<int> >("deque");
    basicOps<list<int> >("list");
} ///:~

```

第1个函数模板，**print()**，演示了能够从任何序列容器中得到的基本信息：容器是否为空、容器当前大小、容器的最大可能尺寸、起始元素和终止元素等。也可以看到，每一个容器都有成员函数**begin()**和**end()**用以返回迭代器。

函数**basicOps()**检测包括多种构造函数在内的所有其他信息（并且依次调用**print()**）：默认构造函数、拷贝构造函数、数量和初值、起始和终止迭代器。还有一个用于赋值的**operator=**和两种类型的**assign()**成员函数。这两个函数其中之一用来获取数量和初始值，而另一个则用来获取起始迭代器和终止迭代器。

所有的基本容器都是可逆容器，就像使用成员函数**rbegin()**和**rend()**所演示的一样。一个序列容器可以重置它的大小，而且可以用**clear()**删除容器中的全部内容（所有元素）。当调用**resize()**扩展一个序列时，新的元素使用序列内元素类型的默认构造函数，如果它们是内置类型，则使用0作为初始值。

用一个迭代器来指定在任何一个序列容器中想要插入元素的起始位置，可以用**insert()**插入单个元素，或插入具有相同值的一组元素，或者由一组起始和终止迭代器标识的来自其他容器的一组元素。

要用**erase()**清除序列中间的一个元素，使用一个迭代器；要用**erase()**清除序列中间的一组元素，使用一对迭代器。注意，因为仅支持双向迭代器，**list**中所有迭代器都只能通过增1或减1来进行移动。（如果容器为可以产生随机访问迭代器的**vector**或者**deque**，**operator+**和**operator-**可以使迭代器移动更大的距离。）

尽管**list**和**deque**支持**push\_front()**和**pop\_front()**，**vector**却不支持，但3者都支持**push\_back()**和**pop\_back()**。

成员函数**swap()**的命名令人有点疑惑，因为还存在另外一个非成员函数**swap()**算法用以交换两个相同类型的对象的值。成员函数**swap()**在两个容器间交换所有东西（如果这两个容器持有相同类型的对象的话），实际上高效地交换了容器本身。它通过交换各个容器的内容来高效地实现交换，这些容器通常存储的是指针。而非成员函数**swap()**算法通常采用赋值的方式来交换其参数（对于整个容器来说是代价比较高的操作），但是对于标准容器来说，它已经通过模板的特化定制为调用成员函数**swap()**了。还有一个**iter\_swap**算法，使用迭代器来交换同一个容器中的两个元素。

以下部分的内容讨论各种类型的序列容器的特点。

#### 7.4.2 向量

**vector**类模板被有意地设计成看起来像一个快速的数组，因为它具有数组风格的索引方式，而且它还可以动态地进行扩展。**vector**类模板具有非常基本的用途，以至于早在本教材的前面就用一种很基本的方法介绍过，并在前面的例子中经常使用。这一节将更深入地介绍**vector**。

为了达到最高效地进行索引和迭代，**vector**将其存储内容作为一个连续的对象数组来维护。在理解**vector**的行为时有一个关键点，那就是索引和迭代操作非常快，基本上和在一个对象数组上进行索引和迭代一样快。但是，这也意味着除了在最后一个元素之后插入新元素（即增补新元素）外，向**vector**中插入一个对象是不可以接受的操作。另外，当一个**vector**预分配的存储空间用完以后，为维护其连续的对象数组，它必须在另一个地方重新分配大块新的（更大的）存储空间，并把以前已有的对象拷贝到新的存储空间中去。这种方法造成了一些令



人不快的副作用。

#### 1. 已分配存储区溢出的代价

**vector**从获取到某块存储区开始,就好像一直在进行猜测:程序员将计划放多少对象进去。在放入的对象还没有超出初始存储块所能装载的对象数目的时候,所有的操作都进行得飞快。(如果程序员预先知道有多少个对象的话,就可以使用**reserve()**预先分配存储区)。但是,在程序运行过程中,最终将会放入过多的对象(超出该存储区的存储范围,即溢出),这时**vector**会做出如下响应:

- 1) 分配一块新的、更大的存储区。
- 2) 将旧存储区中的对象拷贝到新开辟的存储区中去(使用拷贝构造函数)。
- 3) 销毁旧存储区中所有的对象(为每一个对象调用析构函数)。
- 4) 释放旧存储区的内存。

对于复杂对象,如果经常把**vector**装填得过满的话,系统将会为这些拷贝构造和析构操作的完成付出高昂的代价,这就是为什么**vector**(以及一般的STL容器)被设计成值类型(比如那些容易拷贝的类型)容器的原因。其中也包括指针。

为了观察在填充一个**vector**时会发生什么事情,这儿有一个前面已经提到过的**Noisy**类。它打印出其有关创建、析构、赋值以及拷贝构造的信息:

```
//: C07:Noisy.h
// A class to track various object activities.
#ifndef NOISY_H
#define NOISY_H
#include <iostream>
using std::endl;
using std::cout;
using std::ostream;

class Noisy {
    static long create, assign, copycons, destroy;
    long id;
public:
    Noisy() : id(create++) {
        cout << "d[" << id << "]" << endl;
    }
    Noisy(const Noisy& rv) : id(rv.id) {
        cout << "c[" << id << "]" << endl;
        ++copycons;
    }
    Noisy& operator=(const Noisy& rv) {
        cout << "(" << id << ")=[" << rv.id << "]" << endl;
        id = rv.id;
        ++assign;
        return *this;
    }
    friend bool operator<(const Noisy& lv, const Noisy& rv) {
        return lv.id < rv.id;
    }
    friend bool operator==(const Noisy& lv, const Noisy& rv) {
        return lv.id == rv.id;
    }
    ~Noisy() {
        cout << "~[" << id << "]" << endl;
        ++destroy;
    }
    friend ostream& operator<<(ostream& os, const Noisy& n) {
        return os << n.id;
    }
};
```



```

    }
    friend class NoisyReport;
};

struct NoisyGen {
    Noisy operator()() { return Noisy(); }
};

// A Singleton. Will automatically report the
// statistics as the program terminates:
class NoisyReport {
    static NoisyReport nr;
    NoisyReport() {} // Private constructor
    NoisyReport & operator=(NoisyReport &); // Disallowed
    NoisyReport(const NoisyReport&); // Disallowed
public:
    ~NoisyReport() {
        cout << "\n-----\n"
            << "Noisy creations: " << Noisy::create
            << "\nCopy-Constructions: " << Noisy::copycons
            << "\nAssignments: " << Noisy::assign
            << "\nDestructions: " << Noisy::destroy << endl;
    }
};
#endif // NOISY_H ///:~

//: C07:Noisy.cpp {0}
#include "Noisy.h"
long Noisy::create = 0, Noisy::assign = 0,
    Noisy::copycons = 0, Noisy::destroy = 0;
NoisyReport NoisyReport::nr;
///:~

```

每个**Noisy**对象都有其标识符，并且设置了一些静态**static**变量用来跟踪所有的创建、赋值（使用运算符**operator=**）、拷贝构造和析构操作。使用计数器**create**中的默认构造函数来初始化**id**，而拷贝构造函数和赋值操作符则通过右值取得它们的**id**值。与运算符**operator=**在一起的左值是一个已经初始化了的对象，所以**id**在被右值覆盖重写以前打印出其原来的**id**值。

为了支持诸如排序和查找（它们被某些容器隐含地使用）操作，**Noisy**必须有运算符**operator<**和**operator==**。这仅仅是比较**id**值。输出流**ostream**插入符遵循常用的形式并且仅打印出**id**值。

类型**NoisyGen**的对象是在检测期间产生**Noisy**对象的函数对象（因为有一个**operator()**）。

**NoisyReport**是一个单件对象，<sup>⊖</sup>因为我们仅仅要在程序结束时打印一个报告。它有一个私有的**private**构造函数，故不可能创建另外的**NoisyReport**对象，它不允许赋值和拷贝构造，而且有一个静态的名为**nr**的**NoisyReport**实例。在析构函数中只有可执行语句，它们在程序退出和调用静态的析构函数时执行。该析构函数打印出**Noisy**中的所有**static**静态变量所收集的一些统计信息。

使用**Noisy.h**，下列程序演示了一个**vector**分配的存储区溢出的情形：

```

//: C07:VectorOverflow.cpp {-bor}
// Shows the copy-construction and destruction
// that occurs when a vector must reallocate.
//{L} Noisy
#include <cstdlib>
#include <iostream>

```

⊖ 单件是一种著名的设计模式，将在第10章中深入介绍。

```

#include <string>
#include <vector>
#include "Noisy.h"
using namespace std;

int main(int argc, char* argv[]) {
    int size = 1000;
    if(argc >= 2) size = atoi(argv[1]);
    vector<Noisy> vn;
    Noisy n;
    for(int i = 0; i < size; i++)
        vn.push_back(n);
    cout << "\n cleaning up " << endl;
} ///:~

```

可以使用默认值1000，也可以通过命令行键入读者自己设定的值。

当运行该程序时，将会看到一个默认构造函数调用（为n），然后是很多的拷贝构造函数的调用，接下来是一些析构函数的调用，然后又又是很多的拷贝构造函数的调用，等等。当**vector**用光了（超出）为线性数组分配的空间字节时，它必须（维护线性数组中的所有对象，这是它的工作中最重要的部分）获得一块更大的存储空间并且将原来的内容全部移过去，先拷贝然后销毁原来的对象。可以想到，如果存储了大量巨大而复杂的对象，该过程将会很快地变得令人望而却步。

这个问题有两种解决方案。最好的解决方案是要求程序员事先知道到底需要创建多少个对象。在这种情况下，可以使用**reserve()**来告诉**vector**预分配多大的存储区，这样就避免了所有的拷贝和析构操作，而使得任何事情都可以很快地完成（特别是使用操作符**operator[]**来对对象进行随机访问）。注意，使用**reserve()**预分配存储区与通过给出一个整数作为**vector**构造函数的第1个参数是有区别的；后者将使用元素类型的默认构造函数来初始化被规定的元素个数。

通常情况下，程序员并不知道将会需要多少个对象。如果**vector**的重分配操作使程序执行变得迟缓，可以改用其他序列容器。可以使用链表**list**，但是读者将会看到，**deque**允许在序列的两端快速地插入元素，并且在其扩展存储区的时候不需要拷贝和销毁对象的操作。**deque**也允许使用操作符**operator[]**进行随机访问，但是没有**vector**的操作符**operator[]**执行得那么快。因此，如果在程序的某一处创建所有的对象，并且在另一处随机访问它们，可以先填充一个**deque**，再从该**deque**的基础上创建一个**vector**，用以达到快速索引的目的。不应该按照这种习惯去编程——只要知道有这些问题就行了（也就是说，避免过早地进行性能优化）。

然而，**vector**的内存重分配还会带来更糟的问题。因为**vector**在一个优美简洁的数组中保存它的对象，被**vector**使用的迭代器可以是简单的指针。这很好——在所有的序列容器中，这些指针允许以最快的速度选择和操纵容器内的元素。不论它们是简单的指针，还是一个持有指向其容器内部指针的迭代器对象，考虑当添加了一个额外的对象时，为什么会发生导致**vector**进行重新分配存储区并且将其内容移动到别处去的事情。现在那个迭代器的指针指向了一个未知的地方：

```

//: C07:VectorCoreDump.cpp
// Invalidating an iterator.
#include <iterator>
#include <iostream>
#include <vector>
using namespace std;

int main() {

```

```

vector<int> vi(10, 0);
ostream_iterator<int> out(cout, " ");
vector<int>::iterator i = vi.begin();
*i = 47;
copy(vi.begin(), vi.end(), out);
cout << endl;
// Force it to move memory (could also just add
// enough objects):
vi.resize(vi.capacity() + 1);
// Now i points to wrong memory:
*i = 48; // Access violation
copy(vi.begin(), vi.end(), out); // No change to vi[0]
} ///:~

```

这里举例说明了一个称为迭代器无效 (iterator invalidation) 的概念。某种操作引发了涉及容器底层数据的内部变化，因此在变化之前有效的那些迭代器可能后来都不再有效。如果这个程序正试图打破神秘感，那么在向一个 **vector** 加入多个对象时，请查看一下持有的迭代器的位置。在向 **vector** 中加入元素或者使用操作符 **operator[]** 来代替元素选择以后，需要得到一个新迭代器。如果将这种观察结果与向一个 **vector** 加入新对象所知道的潜在开销结合起来看的话，可以得出下述结论。使用一个 **vector** 的最安全的方法，就是一次性地填入所有的元素（在理想的情况下，首先应该知道到底需要多少个对象），然后在程序的另一处仅仅使用它（不再加入更多的元素）。这也是本教材到目前为止使用 **vector** 的方法。标准 C++ 库文档给出了迭代器无效的容器操作。

在前面各章中那些使用 **vector** 作为“基本”容器的内容中，读者可能已经观察到，也许在所有的情况下不是最佳的选择。这是容器和数据结构理论中的一个基本问题，一般而言——“最佳”选择的改变取决于容器的使用方法。到目前为止，**vector** 作为“最佳”选择的理由是它看起来跟一个数组非常相似，因此采用它使读者感到更熟悉和更容易。但是从现在开始，在选择容器时也应该考虑一下有关的其他问题。

## 2. 插入和删除元素

使用 **vector** 最有效的条件是：

- 1) 在开始时用 **reserve()** 分配了正确数量的存储区，因此 **vector** 绝不再重新分配存储区。
- 2) 仅仅在序列的后端添加或者删除元素。

利用一个迭代器向 **vector** 中间插入或者删除元素是可能的，但是下面的程序却演示了一个糟糕的想法：

```

//: C07:VectorInsertAndErase.cpp {-bor}
// Erasing an element from a vector.
//{L} Noisy
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>
#include "Noisy.h"
using namespace std;

int main() {
    vector<Noisy> v;
    v.reserve(11);
    cout << "11 spaces have been reserved" << endl;
    generate_n(back_inserter(v), 10, NoisyGen());
    ostream_iterator<Noisy> out(cout, " ");
    cout << endl;
}

```



```

copy(v.begin(), v.end(), out);
cout << "Inserting an element:" << endl;
vector<Noisy>::iterator it =
    v.begin() + v.size() / 2; // Middle
v.insert(it, Noisy());
cout << endl;
copy(v.begin(), v.end(), out);
cout << "\nErasing an element:" << endl;
// Cannot use the previous value of it:
it = v.begin() + v.size() / 2;
v.erase(it);
cout << endl;
copy(v.begin(), v.end(), out);
cout << endl;
} ///:~

```

在运行该程序的时候，可以看到，对预分配函数**reserve()**的调用实际上仅仅是分配存储区——并没有调用构造函数。对**generate\_n()**的调用非常频繁：每次对**NoisyGen::operator()**的调用都导致一次构造、一次拷贝构造（加入**vector**）和一个临时对象的析构操作。但是，当一个对象被插入到**vector**的中间位置时，必须向后移动该插入位置后面所有的对象以便维持这个线性数组，而且由于有足够的空间，它通过赋值操作符来实现。（如果**reserve()**的参数是10而不是11，它就必须重新分配存储区。）当从**vector**中删除一个对象时，再次使用赋值操作符将该删除位置后面所有的元素向前移动以覆盖被删除元素的位置。（注意，这就要求赋值操作符可以正确地清理左值。）最后，数组末端那个对象被删除。

### 7.4.3 双端队列

双端队列**deque**容器是一种优化了的、在序列两端对元素进行添加和删除操作的基本序列容器。它也允许适度快速地进行随机访问——就像**vector**一样，它也有一个**operator[]**操作符。然而，它没有**vector**的那种把所有的东西都保存在一块连续的内存块中的约束。**deque**的典型实现是利用多个连续的存储块（同时在一个映射结构中保持对这些块及其顺序的跟踪）。因此，向**deque**的两端添加或删除元素所带来的开销很小。另外，它从不需要在分配新的存储区时复制并销毁所包含的对象（就像**vector**所做的那样），所以在向序列两端添加未知数量的对象时，**deque**远比**vector**更有效率。这意味着，只有在确切知道到底需要多少个对象的时候，**vector**才是最优的选择。另外，在本教材前面的章节所列举的许多使用**vector**和**push\_back()**的程序示例，如果改用**deque**替代的话，可能会更有效率。**deque**的接口和**vector**的接口仅仅有很小的不一致（比如，**deque**拥有**push\_front()**和**pop\_front()**，当使用**vector**的时候就没有），所以将使用**vector**的代码转变为使用**deque**要做的工作是微不足道的。考虑程序**StringVector.cpp**，仅仅需要将程序中所有地方的单词“**vector**”改为“**deque**”，就可以使用**deque**了。下列程序将在**StringVector.cpp**中增加与**vector**操作平行的**deque**操作，并且比较它们的执行时间：

```

//: C07:StringDeque.cpp
// Converted from StringVector.cpp.
#include <cstdint>
#include <ctime>
#include <deque>
#include <fstream>
#include <iostream>
#include <iterator>
#include <sstream>
#include <string>
#include <vector>

```

```

#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
    char* fname = "StringDeque.cpp";
    if(argc > 1) fname = argv[1];
    ifstream in(fname);
    assure(in, fname);
    vector<string> vstrings;
    deque<string> dstrings;
    string line;
    // Time reading into vector:
    clock_t ticks = clock();
    while(getline(in, line))
        vstrings.push_back(line);
    ticks = clock() - ticks;
    cout << "Read into vector: " << ticks << endl;
    // Repeat for deque:
    ifstream in2(fname);
    assure(in2, fname);
    ticks = clock();
    while(getline(in2, line))
        dstrings.push_back(line);
    ticks = clock() - ticks;
    cout << "Read into deque: " << ticks << endl;
    // Now compare indexing:
    ticks = clock();
    for(size_t i = 0; i < vstrings.size(); i++) {
        ostringstream ss;
        ss << i;
        vstrings[i] = ss.str() + ": " + vstrings[i];
    }
    ticks = clock() - ticks;
    cout << "Indexing vector: " << ticks << endl;
    ticks = clock();
    for(size_t j = 0; j < dstrings.size(); j++) {
        ostringstream ss;
        ss << j;
        dstrings[j] = ss.str() + ": " + dstrings[j];
    }
    ticks = clock() - ticks;
    cout << "Indexing deque: " << ticks << endl;
    // Compare iteration
    ofstream tmp1("tmp1.tmp"), tmp2("tmp2.tmp");
    ticks = clock();
    copy(vstrings.begin(), vstrings.end(),
        ostream_iterator<string>(tmp1, "\n"));
    ticks = clock() - ticks;
    cout << "Iterating vector: " << ticks << endl;
    ticks = clock();
    copy(dstrings.begin(), dstrings.end(),
        ostream_iterator<string>(tmp2, "\n"));
    ticks = clock() - ticks;
    cout << "Iterating deque: " << ticks << endl;
} ///:~

```

之所以向**vector**中增加对象时执行效率低下，就是因为存储区的重分配，读者可能期待两者之间产生戏剧性的差别。然而，对于一个有1.7MB的文本文件，由一个编译器编译的程序产生的输出如下（测试结果是被测量操作平台 / 编译器中特殊的时钟滴答声，不是秒数）：

```

Read into vector: 8350
Read into deque: 7690

```

```

Indexing vector: 2360
Indexing deque: 2480
Iterating vector: 2470
Iterating deque: 2410

```

不同的编译器和操作平台几乎都同意这个结果。得到的结果并没有产生什么戏剧性的变化，难道不是吗？这指出了一些重要的问题：

1) 我们（程序员和作者）对此进行典型的最坏猜测，就是在程序中的某些地方有低效的事件发生。

2) 效率来自各种效果的组合。在这里，读进每一行并将其转换为字符串就可以控制上面 **vector** 对 **deque** 的代价对照比较。

3) **string** 类在效率方面的设计相当好。

这并不意味着在不确定的对象数将被存入容器末端的时候不用 **deque** 而用 **vector**。正相反，应该用 **deque**——特别是在调整程序的性能的时候。同时也要注意，引起程序性能方面的问题通常并不是在你认为有问题的地方，了解性能瓶颈确切地点的惟一方法就是进行测试。在本章稍后的内容中，读者将会看到 **vector**、**deque** 和 **list** 之间更“纯的”性能比较。

#### 7.4.4 序列容器间的转换

有时在程序的某一部分需要某一种容器的行为或效率，而在程序的另一部分则需要不同容器的行为或效率。比如，在向容器中添加对象时需要 **deque** 的效率，但是在对这些对象进行索引时又需要 **vector** 的效率。每一个基本序列容器（**vector**、**deque** 和 **list**）都有一个双迭代器的构造函数（指明了在创建一个新的对象时在序列中读取的起始和终止位置），和一个用于将数据读入一个现存的容器中的成员函数 **assign()**，所以可以很容易地将对象从一个序列容器移到另一个序列容器。

下面的例子将对象读入 **deque** 内，然后将其转换到一个 **vector**：

```

//: C07:DequeConversion.cpp {-bor}
// Reading into a Deque, converting to a vector.
//{L} Noisy
#include <algorithm>
#include <cstdlib>
#include <deque>
#include <iostream>
#include <iterator>
#include <vector>
#include "Noisy.h"
using namespace std;

int main(int argc, char* argv[]) {
    int size = 25;
    if(argc >= 2) size = atoi(argv[1]);
    deque<Noisy> d;
    generate_n(back_inserter(d), size, NoisyGen());
    cout << "\n Converting to a vector(1)" << endl;
    vector<Noisy> v1(d.begin(), d.end());
    cout << "\n Converting to a vector(2)" << endl;
    vector<Noisy> v2;
    v2.reserve(d.size());
    v2.assign(d.begin(), d.end());
    cout << "\n Cleanup" << endl;
} ///:~

```

读者可以尝试各种尺寸大小的容器，但是请注意，这其实并没有什么差别——这些对象仅被拷贝构造到新的 **vector** 中去。有趣的是，在构建 **vector** 的时候 **v1** 并不会导致多次内存分配，

不管使用多少元素都是这样。读者可能最初会认为，必须遵循用在**v2**上的过程，预分配内存空间以避免零乱的重新分配，但这没有必要，因为**v1**使用的构造函数早就决定了需要的内存空间。

已配置存储区溢出的代价

与**VectorOverflow.cpp**不同，可以更清楚地看到，在使用**deque**的情况下，当一个存储块发生溢出时会发生什么事情。

```

//: C07:DequeOverflow.cpp {-bor}
// A deque is much more efficient than a vector when
// pushing back a lot of elements, since it doesn't
// require copying and destroying.
//{L} Noisy
#include <cstdlib>
#include <deque>
#include "Noisy.h"
using namespace std;

int main(int argc, char* argv[]) {
    int size = 1000;
    if(argc >= 2) size = atoi(argv[1]);
    deque<Noisy> dn;
    Noisy n;
    for(int i = 0; i < size; i++)
        dn.push_back(n);
    cout << "\n cleaning up " << endl;
} ///:~

```

在“cleaning up”输出出现之前，这里有相对较少的（如果有的话）析构函数调用。因为**deque**在多个块中分配所有的存储区，而不是像**vector**一样使用一个类数组的存储区，它从来不需要移动现存的各个数据块的存储区。（因此，就不会有额外的拷贝构造和析构发生。）**deque**仅仅分配一块新存储区。出于相同的原因，**deque**可以高效率地向序列开始端添加元素，因为如果它用完了存储区，它只需（再一次）为序列的开始端分配一个新的存储块。（然而，用于保存数据块索引信息的存储块却有可能需要重新分配。）可是，在一个**deque**的中间插入元素，可能甚至比**vector**更麻烦（但代价不大）。

因为**deque**聪明的存储管理方式，在向**deque**的两端添加元素以后，现存的迭代器都不会失效，就像在演示中对**vector**所做的那样（见**VectorCoreDump.cpp**）。如果坚持**deque**在以下情况下是最好的——从序列的两端插入或删除元素，合理地快速遍历，以及使用**operator[ ]**进行相当快速的随机访问——读者将会形成良好的编程习惯。

#### 7.4.5 被检查的随机访问

**vector**和**deque**都提供了两个随机访问函数：进行索引操作符（**operator[ ]**），这是读者已经看到过的，以及**at( )**，它检测正被索引的容器的边界，如果超出了边界则抛掷出一个异常。使用**at( )**时代价更高一些：

```

//: C07:IndexingVsAt.cpp
// Comparing "at()" to operator[].
#include <ctime>
#include <deque>
#include <iostream>
#include <vector>
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
    long count = 1000;

```



```

int sz = 1000;
if(argc >= 2) count = atoi(argv[1]);
if(argc >= 3) sz = atoi(argv[2]);
vector<int> vi(sz);
clock_t ticks = clock();
for(int i1 = 0; i1 < count; i1++)
    for(int j = 0; j < sz; j++)
        vi[j];
cout << "vector[] " << clock() - ticks << endl;
ticks = clock();
for(int i2 = 0; i2 < count; i2++)
    for(int j = 0; j < sz; j++)
        vi.at(j);
cout << "vector::at() " << clock()-ticks << endl;
deque<int> di(sz);
ticks = clock();
for(int i3 = 0; i3 < count; i3++)
    for(int j = 0; j < sz; j++)
        di[j];
cout << "deque[] " << clock() - ticks << endl;
ticks = clock();
for(int i4 = 0; i4 < count; i4++)
    for(int j = 0; j < sz; j++)
        di.at(j);
cout << "deque::at() " << clock()-ticks << endl;
// Demonstrate at() when you go out of bounds:
try {
    di.at(vi.size() + 1);
} catch(...) {
    cerr << "Exception thrown" << endl;
}
} ///:~

```

就像在第1章中看到的那样，不同的系统采用不同的方法来处理未捕获的异常，但是在使用**at()**的时候，你可以通过多种途径知道程序的某一部分发生了错误，可是在使用**operator[]**时却可能对此一无所知。

#### 7.4.6 链表

**list**以一个双向链表数据结构来实现，如此设计是为了在一个序列的任何地方快速地插入或删除元素，对于**vector**和**deque**而言这是一个代价高得多的操作。**list**没有操作符**operator[]**，所以当对一个**list**进行随机访问时速度非常之慢。其最适用的场合就是在按顺序从头到尾（反之亦然）遍历一个序列的时候，而不是随机地从序列中间选择某一个元素。尽管那样，其遍历速度与**vector**相比仍然较慢，但如果不做那么多遍历操作的话，那就不会成为影响程序性能的瓶颈。

在**list**中每个链接的内存开销需要为实际对象所在的存储区顶部设置一个前向和反向指针。因此，在有较大的对象需要从**list**中间进行插入或删除时，**list**是一个较好的选择。

如果想查找对象要频繁地遍历序列，最好不使用**list**，因为遍历是从**list**的开始端——这是惟一能够开始的地方，除非已经得到一个迭代器，它指向已知道的离目标最近的位置——直到找到感兴趣的那个对象，所耗费的时间与该对象到**list**开始端之间的对象数目成比例。

**list**中的对象在创建以后绝不会移动。“移动”一个**list**的元素意味着改变其链接关系，但绝不会进行拷贝或者对某个实际的对象赋值。这就意味着，在元素被添加到**list**中时，迭代器不会失效，就像较早示例中利用**vector**演示的那样。这里有一个使用**Noisy**对象的**list**的例子：

```

//: C07:ListStability.cpp {-bor}
// Things don't move around in lists.
//{L} Noisy
#include <algorithm>
#include <iostream>
#include <iterator>
#include <list>
#include "Noisy.h"
using namespace std;

int main() {
    list<Noisy> l;
    ostream_iterator<Noisy> out(cout, " ");
    generate_n(back_inserter(l), 25, NoisyGen());
    cout << "\n Printing the list:" << endl;
    copy(l.begin(), l.end(), out);
    cout << "\n Reversing the list:" << endl;
    l.reverse();
    copy(l.begin(), l.end(), out);
    cout << "\n Sorting the list:" << endl;
    l.sort();
    copy(l.begin(), l.end(), out);
    cout << "\n Swapping two elements:" << endl;
    list<Noisy>::iterator it1, it2;
    it1 = it2 = l.begin();
    ++it2;
    swap(*it1, *it2);
    cout << endl;
    copy(l.begin(), l.end(), out);
    cout << "\n Using generic reverse(): " << endl;
    reverse(l.begin(), l.end());
    cout << endl;
    copy(l.begin(), l.end(), out);
    cout << "\n Cleanup" << endl;
} ///:~

```

对于**list**，例如，像进行逆转和排序这些看起来激进的操作都不需要拷贝对象，那是因为，仅需要改变链接而不是移动对象。然而，要注意**sort()**和**reverse()**都是**list**的成员函数，所以它们有**list**内在的特殊知识，能够以再排列元素来代替拷贝它们。另一方面，函数**swap()**则是一个通用算法，它并不了解有关**list**的特别之处，所以它利用拷贝的方法来进行两个元素的交换。一般情况下，如果系统提供了某个算法的成员版本就使用这个成员版本而不使用其等价的通用算法。特别应当指出，通用的**sort()**和**reverse()**算法仅适用于数组、**vector**和**deque**。

如果有较大、复杂的对象，就可能要首先选择**list**，特别是如果构造、析构、拷贝构造以及赋值操作的代价巨大，如果要进行大量的像对对象进行排序或以别的方式对它们进行重新排列操作的时候更是这样。

### 1. 特殊的list操作

**list**有一些特殊的内置操作使其以最好的方式利用**list**的结构。读者已经看到了**reverse()**和**sort()**，这里还有另外一些操作：

```

//: C07:ListSpecialFunctions.cpp
//{L} Noisy
#include <algorithm>
#include <iostream>
#include <iterator>
#include <list>
#include "Noisy.h"
#include "PrintContainer.h"
using namespace std;

```

```

int main() {
    typedef list<Noisy> LN;
    LN l1, l2, l3, l4;
    generate_n(back_inserter(l1), 6, NoisyGen());
    generate_n(back_inserter(l2), 6, NoisyGen());
    generate_n(back_inserter(l3), 6, NoisyGen());
    generate_n(back_inserter(l4), 6, NoisyGen());
    print(l1, "l1", " "); print(l2, "l2", " ");
    print(l3, "l3", " "); print(l4, "l4", " ");
    LN::iterator it1 = l1.begin();
    ++it1; ++it1; ++it1;
    l1.splice(it1, l2);
    print(l1, "l1 after splice(it1, l2)", " ");
    print(l2, "l2 after splice(it1, l2)", " ");
    LN::iterator it2 = l3.begin();
    ++it2; ++it2; ++it2;
    l1.splice(it1, l3, it2);
    print(l1, "l1 after splice(it1, l3, it2)", " ");
    LN::iterator it3 = l4.begin(), it4 = l4.end();
    ++it3; --it4;
    l1.splice(it1, l4, it3, it4);
    print(l1, "l1 after splice(it1,l4,it3,it4)", " ");
    Noisy n;
    LN l5(3, n);
    generate_n(back_inserter(l5), 4, NoisyGen());
    l5.push_back(n);
    print(l5, "l5 before remove()", " ");
    l5.remove(l5.front());
    print(l5, "l5 after remove()", " ");
    l1.sort(); l5.sort();
    l5.merge(l1);
    print(l5, "l5 after l5.merge(l1)", " ");
    cout << "\n Cleanup" << endl;
} ///:~

```

在用**Noisy**对象填充了4个**list**之后，一个**list**通过3种方式结合成另一个**list**。首先，整个链表**l2**在迭代器**it1**处被接合为链表**l1**。注意，在接合之后**l2**是空的——该接合意味着从源链表中删除所有对象。第2次接合从链表**l3**中在迭代器**it2**位置开始，将那些元素插入到链表**l1**中从迭代器**it1**处开始的位置。第3次接合从链表**l1**在迭代器**it1**处开始，并且使用了链表**l4**中始于迭代器**it3**终于迭代器**it4**的那些元素。从表面上看对源链表的提及是多余的，这是因为必须将那些将被传输到目的链表的元素从源链表中删除。

演示删除函数**remove()**的代码输出表明，删除具有特定值的所有元素，链表不必排序。

最后，如果要用**merge()**合并两个链表，只有确定这些链表是否都已经排过序，合并才有意义。在这种情况下最终得到的是一个包含了两个链表中所有元素并排过序的新链表（源链表已经被删除——即其所有的元素已经被移动到目的链表中去了）。

一个**unique()**成员函数将从**list**中删除所有重复的对象，惟一的条件就是首先对**list**进行排序：

```

//: C07:UniqueList.cpp
// Testing list's unique() function.
#include <iostream>
#include <iterator>
#include <list>
using namespace std;
int a[] = { 1, 3, 1, 4, 1, 5, 1, 6, 1 };
const int ASZ = sizeof a / sizeof *a;
int main() {

```

```

// For output:
ostream_iterator<int> out(cout, " ");
list<int> li(a, a + ASZ);
li.unique();
// Oops! No duplicates removed:
copy(li.begin(), li.end(), out);
cout << endl;
// Must sort it first:
li.sort();
copy(li.begin(), li.end(), out);
cout << endl;
// Now unique() will have an effect:
li.unique();
copy(li.begin(), li.end(), out);
cout << endl;
} ///:~

```

在这里使用的**list**构造函数采用来自另外一个容器的起始和超越末尾的迭代器，并将那个容器中的所有元素复制到你自己的**list**中。这里，“容器”仅仅是一个数组，而“迭代器”是指向那个数组的指针，但是因为STL的设计，**list**的构造函数可以像使用任何其他容器一样很容易地使用数组。

函数**unique()**仅仅将毗连的重复元素删除，因此在调用**unique()**之前需要对序列进行排序。当试图解决的问题为根据当前排列的顺序除去毗连的重复元素的时候是个例外。

在这里还有4个附加的**list**成员函数未被演示：**remove\_if()**获得一个预报，该预报决定了某个元素是否能被删除；**unique()**获得一个二元的预报以进行惟一性比较；**merge()**获得一个附加的用于进行比较的参数；**sort()**获得一个比较器（以提供比较或者覆盖当前存在的那个元素）。

## 2. 链表与集合

看一看前面的那个例子，读者可能已经注意到，如果需要一个没有重复元素并且已排过序的序列，得到结果可以是一个集合**set**。对这两个容器的性能进行比较将会很有帮助：

```

//: C07:ListVsSet.cpp
// Comparing list and set performance.
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iterator>
#include <list>
#include <set>
#include "PrintContainer.h"
using namespace std;

class Obj {
    int a[20]; // To take up extra space
    int val;
public:
    Obj() : val(rand() % 500) {}
    friend bool
    operator<(const Obj& a, const Obj& b) {
        return a.val < b.val;
    }
    friend bool
    operator==(const Obj& a, const Obj& b) {
        return a.val == b.val;
    }
    friend ostream&
    operator<<(ostream& os, const Obj& a) {

```



```

        return os << a.val;
    }
};

struct ObjGen {
    Obj operator()() { return Obj(); }
};

int main() {
    const int SZ = 5000;
    srand(time(0));
    list<Obj> lo;
    clock_t ticks = clock();
    generate_n(back_inserter(lo), SZ, ObjGen());
    lo.sort();
    lo.unique();
    cout << "list:" << clock() - ticks << endl;
    set<Obj> so;
    ticks = clock();
    generate_n(inserter(so, so.begin()),
        SZ, ObjGen());
    cout << "set:" << clock() - ticks << endl;
    print(lo);
    print(so);
} ///:~

```

当运行程序的时候，读者会发现**set**比**list**快得多。这是可靠的——毕竟，**set**的主要工作就是在排过序的序列中只保存独一无二的元素。

这个程序使用了头文件**PrintContainer.h**，其中包含一个函数模板，该函数模板用于将任何序列容器打印到一个输出流。**PrintContainer.h**定义如下：

```

//: C07:PrintContainer.h
// Prints a sequence container
#ifndef PRINT_CONTAINER_H
#define PRINT_CONTAINER_H
#include "../C06/PrintSequence.h"

template<class Cont>
void print(Cont& c, const char* nm = "",
           const char* sep = "\n",
           std::ostream& os = std::cout) {
    print(c.begin(), c.end(), nm, sep, os);
}
#endif ///:~

```

这里定义的模板**print()**仅调用了在前一章里的**PrintSequence.h**中定义的函数模板**print()**。

#### 7.4.7 交换序列

我们在前面已经提到过，所有的基本序列容器都有一个成员函数**swap()**，该函数被设计用来将一个序列转换为另一个序列（但只能用于相同类型的序列）。成员函数**swap()**利用了它对特定容器内部结构的知识，从而提高了操作的效率。

```

//: C07:Swapping.cpp {-bor}
// All basic sequence containers can be swapped.
//{L} Noisy
#include <algorithm>
#include <deque>
#include <iostream>
#include <iterator>
#include <list>

```

```

#include <vector>
#include "Noisy.h"
#include "PrintContainer.h"
using namespace std;
ostream_iterator<Noisy> out(cout, " ");

template<class Cont> void testSwap(char* cname) {
    Cont c1, c2;
    generate_n(back_inserter(c1), 10, NoisyGen());
    generate_n(back_inserter(c2), 5, NoisyGen());
    cout << endl << cname << ":" << endl;
    print(c1, "c1"); print(c2, "c2");
    cout << "\n Swapping the " << cname << ":" << endl;
    c1.swap(c2);
    print(c1, "c1"); print(c2, "c2");
}

int main() {
    testSwap<vector<Noisy> >("vector");
    testSwap<deque<Noisy> >("deque");
    testSwap<list<Noisy> >("list");
} ///:~

```

在运行这个程序时，读者将会发现，每一种类型的序列容器都能在不需要复制或者赋值的情况下将一种序列变换为另一种序列，即使这些序列的尺寸不同。实际上，其所做的是完全地将一个对象的资源交换为另一个对象的资源。

STL算法也包含一个**swap()**，当该函数应用于两个相同类型的容器时，它使用成员函数**swap()**来达到快速的性能。所以，如果对容器的一个容器应用**sort()**算法，读者会发现其执行速度非常快——这表明对容器的一个容器进行快速排序也是设计STL的一个目的。

## 7.5 集合

集合(**set**)容器仅接受每个元素的一个副本。它也对元素排序。(进行排序并不是**set**的概念定义所固有的，但是STL **set**用一棵平衡树数据结构来存储其元素以提供快速的查找，因此在遍历**set**的时候就产生了排序的结果。)在本章的前两个例子中用到了**set**。

考虑为一本书创建索引的问题。有人可能喜欢从书中的所有单词开始创建，但是每个单词只需要一个实例，并且希望它们排过序。对于这个问题，容器**set**是理想的选择，它可以毫不费力地解决这个问题。然而，还存在标点符号和非字母字符的问题，它们必须被去掉以便产生正确的单词。该问题的一个解决方案就是用标准C库函数**isalpha()**和**isspace()**提取只需要的字符。可以用空白字符来替换所有不需要的字符，这样就可以很容易地从读入的每一行中提取出合法的单词：

```

//: C07:WordList.cpp
// Display a list of words used in a document.
#include <algorithm>
#include <cctype>
#include <cstring>
#include <fstream>
#include <iostream>
#include <iterator>
#include <set>
#include <sstream>
#include <string>
#include "../require.h"
using namespace std;

```

```

char replaceJunk(char c) {
    // Only keep alphas, space (as a delimiter), and '
    return (isalpha(c) || c == '\\') ? c : ' ';
}

int main(int argc, char* argv[]) {
    char* fname = "WordList.cpp";
    if(argc > 1) fname = argv[1];
    ifstream in(fname);
    assure(in, fname);
    set<string> wordlist;
    string line;
    while(getline(in, line)) {
        transform(line.begin(), line.end(), line.begin(),
            replaceJunk);
        istringstream is(line);
        string word;
        while(is >> word)
            wordlist.insert(word);
    }
    // Output results:
    copy(wordlist.begin(), wordlist.end(),
        ostream_iterator<string>(cout, "\n"));
} ///:~

```

调用**transform()**，以空白字符替换每个要被忽略掉的字符。容器**set**不但忽略重复的单词，而且根据函数对象**less<string>**（**set**容器默认的第2个模板参数）比较它保存的那些单词，该函数依次使用**string::operator<()**进行比较操作，因此这些单词按字母顺序出现。

仅仅为了得到一个经过排序的序列，就没有必要使用**set**。可以在不同的STL容器上使用函数**sort()**（与STL众多的其他函数）来达到这个目的。然而，在这里或许**set**将会更快地完成该操作。当只想做查找操作时，使用**set**将会特别便利，因为其**find()**成员函数有对数级的复杂性，因此它比通用的**find()**算法要快得多。如前所述，通用的**find()**算法需要遍历全部范围直到找到要搜寻的元素（这将导致最坏情况下算法复杂性为 $N$ ，平均复杂性为 $N/2$ ）。然而，如果有一个已经排过序的序列容器，在查找元素时使用**equal\_range()**，就可以得到对数级的算法复杂性。

下面这个程序显示了如何使用迭代器**istreambuf\_iterator**来构建单词表，迭代器将字符从一个地方（输入流）移到另一个地方（一个**string**对象），该操作依赖标准C库函数**isalpha()**的返回值是否为真：

```

//: C07:WordList2.cpp
// Illustrates istreambuf_iterator and insert iterators.
#include <cstring>
#include <fstream>
#include <iostream>
#include <iterator>
#include <set>
#include <string>
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
    char* fname = "WordList2.cpp";
    if(argc > 1) fname = argv[1];
    ifstream in(fname);
    assure(in, fname);
    istreambuf_iterator<char> p(in), end;
    set<string> wordlist;
    while(p != end) {

```



```

    string word;
    insert_iterator<string> ii(word, word.begin());
    // Find the first alpha character:
    while(p != end && !isalpha(*p))
        ++p;
    // Copy until the first non-alpha character:
    while(p != end && isalpha(*p))
        *ii++ = *p++;
    if(word.size() != 0)
        wordlist.insert(word);
}
// Output results:
copy(wordlist.begin(), wordlist.end(),
    ostream_iterator<string>(cout, "\n"));
} ///:~

```

这个例子是由**Nathan Myers**提议的，他发明了**istreambuf\_iterator**及其家族成员。这个迭代器从一个流中逐字符地提取信息。虽然**istreambuf\_iterator**的模板参数可能暗示读者提取例如以**int**型数据而非**char**型数据，但事实却不是这样。该参数必须是某些字符类型——常规的**char**类型或宽字符类型。

文件打开以后，称为**p**的**istreambuf\_iterator**被附到该**istream**上，这样就可以从中提取字符了。名为**wordlist**的**set<string>**将保存作为结果的单词。

采用**while**循环从输入流中读取单词直到发现输入流结束。这是用**istreambuf\_iterator**的默认构造函数进行检测的，它产生超越末尾的迭代器对象**end**。因此，如果要进行测试以确信不在该流的末尾，只需用**p!=end**。

在这里使用的第2个迭代器类型是在前面已经看到的**insert\_iterator**。利用它将对象插入一个容器。在这里，“容器”是一个称为**word**的**string**，它对于**insert\_iterator**的目的来说，其行为像一个容器。对于**insert\_iterator**的构造函数，则需要该容器和一个指明应在何处开始插入这些字符的迭代器。也可以使用一个**back\_insert\_iterator**，它需要容器拥有一个**push\_back()**（**string**产生）。

**while**循环在做好各种准备后，就开始查找第1个字母字符，对**start**进行增1操作直到那个字符被找到。然后将查找到的字符从一个迭代器指向的位置复制到另一个迭代器指向的位置，当发现一个非字母字符时停止复制。假定其不为空，则每一个**word**都被添加到**wordlist**中去。

### 可完全重用的标识符识别器

单词表例子使用不同的方法从流中提取标识符，但它们都不特别灵活。因为STL容器和算法都是围绕着迭代器来展开的，所以最灵活的解决方法是它自己使用迭代器。可以把**TokenIterator**想象成一个迭代器，该迭代器封装其任何能够产生字符的其他迭代器。因为它确实是一个输入迭代器类型（最原始的迭代器类型），可以向任何STL算法提供输入。不仅其本身就是一个很有用的工具，下列的**TokenIterator**也是如何设计用户自己的迭代器的很好的例子。<sup>①</sup>

**TokenIterator**类具有双重灵活性。首先，可以选择产生**char**输入的迭代器类型。其次，**TokenIterator**不是仅说明什么字符表示分界符，而是使用一个函数对象判定函数，其**operator()**接受一个**char**型参数并决定它是否将计入标识符。虽然这两个例子在这里给出了什么字符属于标识符的静态概念，但是用户可以很容易地设计自己的函数对象，以便在读入字符的时候改变其状态，创建一个更复杂的解析器。

① 这是Nathan Myers讲授的另一个例子。



和**TokenIterator**模板一起，下列的头文件还包含了两个基本判定函数，**Isalpha**和**Delimiters**：

```
//: C07:TokenIterator.h
#ifndef TOKENITERATOR_H
#define TOKENITERATOR_H
#include <algorithm>
#include <cctype>
#include <functional>
#include <iterator>
#include <string>

struct Isalpha : std::unary_function<char, bool> {
    bool operator()(char c) { return std::isalpha(c); }
};

class Delimiters : std::unary_function<char, bool> {
    std::string exclude;
public:
    Delimiters() {}
    Delimiters(const std::string& excl) : exclude(excl) {}
    bool operator()(char c) {
        return exclude.find(c) == std::string::npos;
    }
};

template<class InputIter, class Pred = Isalpha>
class TokenIterator : public std::iterator<
    std::input_iterator_tag, std::string, std::ptrdiff_t> {
    InputIter first;
    InputIter last;
    std::string word;
    Pred predicate;
public:
    TokenIterator(InputIter begin, InputIter end,
        Pred pred = Pred())
        : first(begin), last(end), predicate(pred) {
        ++*this;
    }
    TokenIterator() {} // End sentinel
    // Prefix increment:
    TokenIterator& operator++() {
        word.resize(0);
        first = std::find_if(first, last, predicate);
        while(first != last && predicate(*first))
            word += *first++;
        return *this;
    }
    // Postfix increment
    class CaptureState {
        std::string word;
    public:
        CaptureState(const std::string& w) : word(w) {}
        std::string operator*() { return word; }
    };
    CaptureState operator++(int) {
        CaptureState d(word);
        ++*this;
        return d;
    }
    // Produce the actual value:
    std::string operator*() const { return word; }
    const std::string* operator->() const { return &word; }
    // Compare iterators:
```



```

bool operator==(const TokenIterator&) {
    return word.size() == 0 && first == last;
}
bool operator!=(const TokenIterator& rv) {
    return !(*this == rv);
}
};
#endif // TOKENITERATOR_H ///:~

```

**TokenIterator**类派生自**std::iterator**模板。它可能表现出某些来自**std::iterator**的功能，但它是纯粹对迭代器进行标记的一种方式，用来告诉使用它的容器可以做什么。在这里，可以看到作为**iterator\_category**模板参数的**input\_iterator\_tag**——这告诉那些询问者，**TokenIterator**只有一个输入迭代器的能力，不能与那些需要更复杂的迭代器的算法一起使用。除了进行标记以外，**std::iterator**不能做超出提供几个有用的类型定义的任何事情。用户必须自行实现所有其他的功能。

起初读者可能会认为**TokenIterator**类有点奇怪，因为其第1个构造函数需要“开始”和“终止”两个迭代器作为参数，和它们在一起的还有一个判定函数。记住，这是一个“封装器”迭代器，在输入结束时它没有办法如何告知何时其输入处于末尾，所以在第1个构造函数中这个“终止”迭代器是必需的。第2个（默认）构造函数存在的理由是，STL算法（以及所有用户自己编写的算法）需要一个**TokenIterator**标记作为超越末尾的值。因为判断一个**TokenIterator**是否已经到达其输入的末尾的所有信息都已经由第1个构造函数收集，这第2个构造函数创建**TokenIterator**对象，在算法中它只作为占位符使用。

行为的核心发生在运算符**operator++**中。它使用**string::resize()**擦除当前**word**的值，然后使用**find\_if()**寻找第1个满足判定函数的字符（如此来发现一个新的标识符的起始位置）。结果迭代器被分配给**first**，因此将**first**向前移动至标识符的起始位置。然后，一旦判定函数被满足而又没有到达输入的末尾，输入字符就被复制到**word**中。最后，**TokenIterator**对象返回，并且必须被解析以便访问新的标识符。

这里的前缀增1要求有一个**CaptureState**类型的对象在增1前持有值，因此它是可以返回的。产生的实际值是一个直接的**operator\***操作。为输出迭代器定义的其余的函数仅仅是**operator==**和**operator!=**，用以指明**TokenIterator**是否已经到达了其输入的末尾。读者可以看到**operator==**的参数被忽略——它仅仅关心是否已经到达其内部的**last**迭代器。注意，**operator!=**是通过**operator==**定义的。

一个好的**TokenIterator**测试包括许多不同来源的输入字符，包括一个**streambuf\_iterator**、一个**char\***和一个**deque<char>::iterator**。最后，最初的单词表的问题解决如下：

```

//: C07:TokenIteratorTest.cpp {-g++}
#include <fstream>
#include <iostream>
#include <vector>
#include <deque>
#include <set>
#include "TokenIterator.h"
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
    char* fname = "TokenIteratorTest.cpp";
    if(argc > 1) fname = argv[1];
    ifstream in(fname);
}

```

```

    assure(in, fname);
    ostream_iterator<string> out(cout, "\n");
    typedef istreambuf_iterator<char> IsbIt;
    IsbIt begin(in), isbEnd;
    Delimiters delimiters(" \\t\\n~;()\\\"<>:{ }[]+-=&*#./\\");
    TokenIterator<IsbIt, Delimiters>
        wordIter(begin, isbEnd, delimiters), end;
    vector<string> wordlist;
    copy(wordIter, end, back_inserter(wordlist));
    // Output results:
    copy(wordlist.begin(), wordlist.end(), out);
    *out++ = "-----";
    // Use a char array as the source:
    char* cp = "typedef std::istreambuf_iterator<char> It";
    TokenIterator<char*, Delimiters>
        charIter(cp, cp + strlen(cp), delimiters), end2;
    vector<string> wordlist2;
    copy(charIter, end2, back_inserter(wordlist2));
    copy(wordlist2.begin(), wordlist2.end(), out);
    *out++ = "-----";
    // Use a deque<char> as the source:
    ifstream in2("TokenIteratorTest.cpp");
    deque<char> dc;
    copy(IsbIt(in2), IsbIt(), back_inserter(dc));
    TokenIterator<deque<char>::iterator, Delimiters>
        dcIter(dc.begin(), dc.end(), delimiters), end3;
    vector<string> wordlist3;
    copy(dcIter, end3, back_inserter(wordlist3));
    copy(wordlist3.begin(), wordlist3.end(), out);
    *out++ = "-----";
    // Reproduce the Wordlist.cpp example:
    ifstream in3("TokenIteratorTest.cpp");
    TokenIterator<IsbIt, Delimiters>
        wordIter2(IsbIt(in3), isbEnd, delimiters);
    set<string> wordlist4;
    while(wordIter2 != end)
        wordlist4.insert(*wordIter2++);
    copy(wordlist4.begin(), wordlist4.end(), out);
} ///:~

```

在使用**istreambuf\_iterator**时，创建一个附属于**istream**的对象，并与默认构造函数一起作为超越末尾标记。这两者用于创建将产生标识符的**TokenIterator**，而默认构造函数创建一个伪**TokenIterator**超越末尾的标记。（这仅仅是个占位符并且被忽略。）**TokenIterator**产生那些要被插入**string**容器的**string**——在这里，除了最后一个以外，在所有的情况下都使用**vector<string>**。（也可以将所有的结果连接成一个**string**。）除此之外，**TokenIterator**就像任何其他输入迭代器一样工作。

在定义一个双向（并且因此也成为随机访问）迭代器时，可以使用**std::reverse\_iterator**适配器“免费地”得到反向迭代器。如果已经为一个具有双向能力的容器定义了一个迭代器的话，可以从如下在容器类里的前向遍历迭代器那里得到一个反向迭代器：

```

// Assume "iterator" is your nested iterator type
typedef std::reverse_iterator<iterator> reverse_iterator;
reverse_iterator rbegin() {return reverse_iterator(end());}
reverse_iterator rend() {return reverse_iterator(begin());}

```

**std::reverse\_iterator**适配器可以做所有这些工作。比如，如果使用运算符\*来解析反向迭代器，它自动地对它持有的前向迭代器的一个临时拷贝减1，以便返回正确的元素，因为反向迭代器在逻辑上指向它们引用的元素的下一个位置。

## 7.6 堆栈

堆栈`stack`容器，与`queue`和`priority_queue`一起被归类为适配器，这意味着它们将通过调整某一个基本序列容器以存储自己的数据。这是一个遗憾的令人困惑的情况，为什么某些事情一定要与它的底层实现的细节联系在一起呢——这些容器被称为“适配器”的真相只对库的创建者才有基本的价值。当用户使用它们时，通常并不关心它们是否是适配器，仅需知道它们能够解决自己的问题就行了。诚然，有时知道可以选择不同的实现或者在现存的容器对象之上建立一个适配器是很有用的，但是，通常那一层次的功能已经在适配器的行为中被删除了。因此，如果看到在别处某个容器被强调是一个适配器，一般只能指出实际上什么时候它是有用的。注意，每一种类型的适配器都有一个该适配器构建在其上的默认的容器，而且这种默认是最明智的实现方式。在大多数情况下，用户不必关心容器的底层的具体实现。

下面的例子显示实现`stack<string>`的3种方式：默认方式（使用`deque`），然后采用`vector`的方式，最后一个采用`list`的方式：

```
//: C07:Stack1.cpp
// Demonstrates the STL stack.
#include <fstream>
#include <iostream>
#include <list>
#include <stack>
#include <string>
#include <vector>
using namespace std;

// Rearrange comments below to use different versions.
typedef stack<string> Stack1; // Default: deque<string>
// typedef stack<string, vector<string> > Stack2;
// typedef stack<string, list<string> > Stack3;

int main() {
    ifstream in("Stack1.cpp");
    Stack1 textlines; // Try the different versions
    // Read file and store lines in the stack:
    string line;
    while(getline(in, line))
        textlines.push(line + "\n");
    // Print lines from the stack and pop them:
    while(!textlines.empty()) {
        cout << textlines.top();
        textlines.pop();
    }
} ///:~
```

如果读者使用过其他`stack`类的话，这里的`top()`和`pop()`操作似乎并不直观。当调用`pop()`时，它返回一个`void`值而不是所预期的栈顶元素。如果想要栈顶元素，可以通过`top()`取得指向它的一个引用。这样做的结果效率更高，因为传统的`pop()`函数必须返回一个值而不是一个引用，因此调用拷贝构造函数。更重要的是，这是异常安全的（**exception safe**），就像我们在第1章中讨论的那样。如果`pop()`在改变栈状态的同时试图返回栈顶元素，那么在元素的拷贝构造函数中产生的某个异常就会导致元素的丢失。在使用`stack`（或者一个`priority_queue`，将在稍后描述）时，可以高效地查阅`top()`，就像你希望得那么快，然后明确使用`pop()`将栈顶元素丢弃。（也许，如果使用一些不同于大家熟悉的“**pop**”这样的术语来定义函数，解释起来可能会更清楚一点儿。）

**stack**模板有一个简单的接口——本质上就是在较早前看到的那些成员函数。因为对于一个**stack**来说，只有访问其栈顶元素才有意义，没有提供能够遍历它的迭代器。也没有复杂的初始化形式，但是如果需要这样做的话，可以使用**stack**的底层容器。比如，假定有一个函数，期望**stack**的接口，但是在程序的其余部分需要将对象存储在**list**中。下面的程序存储文件中的每一行，与该行中的前导空白字符的个数一起存储。（可以想象把它作为对源代码执行某种重新格式化操作的出发点。）

```
//: C07:Stack2.cpp
// Converting a list to a stack.
#include <iostream>
#include <fstream>
#include <stack>
#include <list>
#include <string>
#include <cstdint>
using namespace std;

// Expects a stack:
template<class Stk>
void stackOut(Stk& s, ostream& os = cout) {
    while(!s.empty()) {
        os << s.top() << "\n";
        s.pop();
    }
}

class Line {
    string line; // Without leading spaces
    size_t lspaces; // Number of leading spaces
public:
    Line(string s) : line(s) {
        lspaces = line.find_first_not_of(' ');
        if(lspaces == string::npos)
            lspaces = 0;
        line = line.substr(lspaces);
    }
    friend ostream& operator<<(ostream& os, const Line& l) {
        for(size_t i = 0; i < l.lspaces; i++)
            os << ' ';
        return os << l.line;
    }
    // Other functions here...
};

int main() {
    ifstream in("Stack2.cpp");
    list<Line> lines;
    // Read file and store lines in the list:
    string s;
    while(getline(in, s))
        lines.push_front(s);
    // Turn the list into a stack for printing:
    stack<Line, list<Line> > stk(lines);
    stackOut(stk);
} ///:~
```

需要**stack**接口的函数仅仅发送每个**top()**对象到一个**ostream**，然后通过调用**pop()**将其删除。**Line**类判断前导空白字符的个数，然后存储没有这些前导空白字符的行内容。**ostream operator<<**重新插入前导空白字符，因此该行能够被正确地打印出来，但是能很容易地通过改变**lspaces**的值来改变空白字符的个数。（做这件事的成员函数没有在这里显示。）

在`main()`函数中,输入文件被读入到`list<Line>`,然后链表中的每一行都被复制到一个`stack`,该`stack`被送到`stackOut()`函数中。

不能从头至尾对一个`stack`进行迭代;这就强调了,当创建一个`stack`时,只能希望对其进行`stack`操作。可以使用一个`vector`及其`back()`、`push_back()`和`pop_back()`成员函数获得等价的“堆栈”功能,还拥有`vector`的所有附加的功能。程序`Stack1.cpp`可以重写成如下形式:

```
//: C07:Stack3.cpp
// Using a vector as a stack; modified Stack1.cpp.
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main() {
    ifstream in("Stack3.cpp");
    vector<string> textlines;
    string line;
    while(getline(in, line))
        textlines.push_back(line + "\n");
    while(!textlines.empty()) {
        cout << textlines.back();
        textlines.pop_back();
    }
} ///:~
```

这个程序产生像`Stack1.cpp`一样输出,但现在还可以进行与`vector`一样的操作。`list`也可以将元素压入前端,但是它通常比与`vector`一起使用`push_back()`的效率低。(另外,对于将元素压入前端的操作,`deque`通常比`list`的效率更高。)

## 7.7 队列

`queue`容器是一个受到限制的`deque`形式——只可以在队列一端放入元素,而在另一端删除它们。在功能上,可以在需要使用`queue`的任何地方使用`deque`,那时也能够使用`deque`的附加功能。需要使用`queue`而不是`deque`的惟一理由就是,当读者希望强调仅仅执行与`queue`相似的行为的时候。

`queue`类是一个如同`stack`的适配器,因为它也建立在另一个序列容器的基础之上。就像读者猜测的那样,对`queue`的理想实现是`deque`,而其对`queue`来说是默认的模板参数;很少需要不同的实现。

如果想建立这样一个系统模型,即系统中的某些元素正在等待另一些元素的服务时,时常使用队列。“银行出纳员问题”就是一个经典的例子。顾客们在随机的时间间隔到达银行,进入某一行队列排队,然后由一组出纳员服务。因为顾客们到达是随机的,并且每一个顾客得到的服务时间总数也是随机的,所以没有一种方法能决定性地知道在任何时间点某行队列有多长。然而,模拟这种情形并且看看到底会发生什么事情却是可能的。

在对现实的模拟中,每个顾客和每个出纳员都在独立的线程中运行。这多么像是一个多线程的环境,因此每个顾客和出纳员都有他自己的线程。然而,标准C++不支持多线程处理。另一方面,通过对代码做一些小的调整,模拟足够的多线程处理以提供一个满意的解决方案是可能的。<sup>①</sup>

① 我们将在第11章再次讨论多线程处理问题。

在多线程处理中，多个受控制的线程同时运行，共享同一个地址空间。通常用户拥有的CPU数量都会比运行的线程数量少（常常只有一个CPU）。为得到虚拟的环境，应使每一个线程都拥有其自己的CPU，一种时间分片（**time-slicing**）机制说“OK，当前线程你已经占用了足够多的时间，我马上就要让你停止从而给其他线程一些时间了”。这种自动的线程停止和启动被称为抢占（**preemptive**），它意味着（程序员）不需要去管理线程处理的过程。

另一种方法就是每个线程自动地将CPU让给线程调度器，该线程调度器然后寻找另一个需要运行的线程。另外，建立“时间分片”进入到系统中的各个类。在这里，将那些出纳员表示为“线程”（顾客将是被动的）。每个出纳员都有一个进行无限循环处理的成员函数**run()**，该成员函数将在执行确定数量的“时间单元”后返回。通过使用平常的返回机制，排除了任何需要进行的交换处理。虽然产生的程序很小，但是它提供了一个不平常的合理的模拟场景：

```
//: C07:BankTeller.cpp {RunByHand}
// Using a queue and simulated multithreading
// to model a bank teller system.
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iterator>
#include <list>
#include <queue>
using namespace std;

class Customer {
    int serviceTime;
public:
    Customer() : serviceTime(0) {}
    Customer(int tm) : serviceTime(tm) {}
    int getTime() { return serviceTime; }
    void setTime(int newtime) { serviceTime = newtime; }
    friend ostream&
    operator<<(ostream& os, const Customer& c) {
        return os << '[' << c.serviceTime << ']'<
    }
};

class Teller {
    queue<Customer>& customers;
    Customer current;
    enum { SLICE = 5 };
    int ttime; // Time left in slice
    bool busy; // Is teller serving a customer?
public:
    Teller(queue<Customer>& cq)
        : customers(cq), ttime(0), busy(false) {}
    Teller& operator=(const Teller& rv) {
        customers = rv.customers;
        current = rv.current;
        ttime = rv.ttime;
        busy = rv.busy;
        return *this;
    }
    bool isBusy() { return busy; }
    void run(bool recursion = false) {
        if(!recursion)
            ttime = SLICE;
        int servtime = current.getTime();
        if(servtime > ttime) {
            servtime -= ttime;

```



```

        current.setTime(servtime);
        busy = true; // Still working on current
        return;
    }
    if(servtime < ttime) {
        ttime -= servtime;
        if(!customers.empty()) {
            current = customers.front();
            customers.pop(); // Remove it
            busy = true;
            run(true); // Recurse
        }
        return;
    }
    if(servtime == ttime) {
        // Done with current, set to empty:
        current = Customer(0);
        busy = false;
        return; // No more time in this slice
    }
}
};

// Inherit to access protected implementation:
class CustomerQ : public queue<Customer> {
public:
    friend ostream&
    operator<<(ostream& os, const CustomerQ& cd) {
        copy(cd.c.begin(), cd.c.end(),
            ostream_iterator<Customer>(os, ""));
        return os;
    }
};

int main() {
    CustomerQ customers;
    list<Teller> tellers;
    typedef list<Teller>::iterator TellIt;
    tellers.push_back(Teller(customers));
    srand(time(0)); // Seed the random number generator
    clock_t ticks = clock();
    // Run simulation for at least 5 seconds:
    while(clock() < ticks + 5 * CLOCKS_PER_SEC) {
        // Add a random number of customers to the
        // queue, with random service times:
        for(int i = 0; i < rand() % 5; i++)
            customers.push(Customer(rand() % 15 + 1));
        cout << '{' << tellers.size() << '}'
            << customers << endl;
        // Have the tellers service the queue:
        for(TellIt i = tellers.begin();
            i != tellers.end(); i++)
            (*i).run();
        cout << '{' << tellers.size() << '}'
            << customers << endl;
        // If line is too long, add another teller:
        if(customers.size() / tellers.size() > 2)
            tellers.push_back(Teller(customers));
        // If line is short enough, remove a teller:
        if(tellers.size() > 1 &&
            customers.size() / tellers.size() < 2)
            for(TellIt i = tellers.begin();
                i != tellers.end(); i++)

```





```

        if(!(*i).isBusy()) {
            tellers.erase(i);
            break; // Out of for loop
        }
    }
} ///:~

```

每个顾客都需要一个确定的服务时间总额，这就是一个出纳员必须在为某个顾客提供其所需服务上花费的时间单元数。为每个顾客提供的服务时间总额都不同，并且这个时间总额肯定是随机的。另外，也不会知道在每个时间间隔内究竟会有多少个顾客到达，因此这也肯定是随机的。

这些顾客**Customer**对象被保存在一个**queue<Customer>**中，并且每个出纳员**Teller**对象都持有那个队列的一个引用。在一个**Teller**对象完成对当前**Customer**对象的服务以后，这个**Teller**将会从队列中得到另一个**Customer**，开始继续为这个新**Customer**提供服务，系统从该**Teller**分配到的时间片里面减少**Customer**的服务时间。所有这些逻辑都包含在成员函数**run()**中，它只是一个具有3个分支的**if**语句，该语句基于以下事件建立，即当前顾客所必需的服务时间总额是小于、大于、或是等于出纳员在当前时间片中剩余的时间总额。注意，如果该出纳员**Teller**在完成对一个**Customer**的服务后还有多余的时间，它再获得一个新的**Customer**，然后实施自身的递归处理。就像使用**stack**一样，在使用**queue**时，它只是一个**queue**，不具有基础序列容器的任何其他功能。这包括获得一个迭代器以遍历**stack**的能力。然而，在**queue**内部将底层序列容器（构建**queue**的基础）作为一个**protected**成员来保存，在C++标准中该成员被指定以'**c**'来做标识符，这意味着可以通过派生自**queue**的类来访问底层实现。在这里类**CustomerQ**正是这样做的，其惟一目的就是定义一个**ostream operator<<**，以便在**queue**上实施迭代并显示其成员。

这个模拟系统的驱动器就是**main()**函数中的**while**循环，它使用处理器的时钟滴答（定义于**<ctime>**中）来决定该模拟系统是否已经至少运行了5秒钟。在每次经过从头到尾的循环的开始，都要加入随机的顾客数，和随机的服务时间。为了看到系统当前的状态，出纳员的数量和队列的内容都将显示出来。每个出纳员处理完后，重复地显示这些信息。在这一点上，系统通过比较顾客和出纳员的数量来进行调整。如果某行队列太长，就加入其他的出纳员，而如果队列足够短，则删除一个出纳员。在程序中的这个调整区段中，可以用实验的策略得到关于添加和删除出纳员的最佳数据。如果这是惟一想要修改的区段，也许要将策略封装到不同的对象中去。

本教材将在第11章中的多线程练习中再次涉及这个例子。

## 7.8 优先队列

当向一个优先队列**priority\_queue**用**push()**压入一个对象时，那个对象根据一个比较函数或函数对象在队列中排序。（可以允许用默认的**less**模板来代替这个函数或函数对象，或者可以提供一个用户自己定义的函数或函数对象。）**priority\_queue**确定在用**top()**查看顶部元素时，该元素将是具有最高优先级的一个元素。当处理完该元素以后，调用**pop()**删除它，并且促使下一个元素进入该位置。因此，**priority\_queue**拥有与**stack**几乎相同的接口，但它的表现不同。

就像**stack**和**queue**一样，**priority\_queue**是一个基于某个基本序列容器进行构建的适配器——默认的序列容器是**vector**。

创建一个用来处理**int**型数据的**priority\_queue**是个很平常的工作：

```

//: C07:PriorityQueue1.cpp
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <queue>
using namespace std;

int main() {
    priority_queue<int> pqi;
    srand(time(0)); // Seed the random number generator
    for(int i = 0; i < 100; i++)
        pqi.push(rand() % 25);
    while(!pqi.empty()) {
        cout << pqi.top() << ' ';
        pqi.pop();
    }
} ///:~

```

该程序向**priority\_queue**压入100个数值介于0到24之间的随机数。在运行这个程序时，会看到它允许出现重复的值，而且最高值先出现。为了演示如何通过提供用户自己的函数或函数对象以改变其元素的排列顺序，下面的程序给予较低值的数以最高的优先级：

```

//: C07:PriorityQueue2.cpp
// Changing the priority.
#include <cstdlib>
#include <ctime>
#include <functional>
#include <iostream>
#include <queue>
using namespace std;

int main() {
    priority_queue<int, vector<int>, greater<int> > pqi;
    srand(time(0));
    for(int i = 0; i < 100; i++)
        pqi.push(rand() % 25);
    while(!pqi.empty()) {
        cout << pqi.top() << ' ';
        pqi.pop();
    }
} ///:~

```

一个更有趣的问题是to-do列表，这里每个对象都包含一个**string**，和一个主优先级及一个次优先级的值：

```

//: C07:PriorityQueue3.cpp
// A more complex use of priority_queue.
#include <iostream>
#include <queue>
#include <string>
using namespace std;

class ToDoItem {
    char primary;
    int secondary;
    string item;
public:
    ToDoItem(string td, char pri = 'A', int sec = 1)
        : primary(pri), secondary(sec), item(td) {}
    friend bool operator<(
        const ToDoItem& x, const ToDoItem& y) {
        if(x.primary > y.primary)

```



```

        return true;
    if(x.primary == y.primary)
        if(x.secondary > y.secondary)
            return true;
        return false;
    }
    friend ostream&
    operator<<(ostream& os, const ToDoItem& td) {
        return os << td.primary << td.secondary
            << ": " << td.item;
    }
};

int main() {
    priority_queue<ToDoItem> toDoList;
    toDoList.push(ToDoItem("Empty trash", 'C', 4));
    toDoList.push(ToDoItem("Feed dog", 'A', 2));
    toDoList.push(ToDoItem("Feed bird", 'B', 7));
    toDoList.push(ToDoItem("Mow lawn", 'C', 3));
    toDoList.push(ToDoItem("Water lawn", 'A', 1));
    toDoList.push(ToDoItem("Feed cat", 'B', 1));
    while(!toDoList.empty()) {
        cout << toDoList.top() << endl;
        toDoList.pop();
    }
} //:~

```

由于是与`less<>`一同工作，所以`ToDoItem`的`operator<`必须是一个非成员函数。除此之外，每一件事情都是自动发生的。输出结果如下：

```

A1: Water lawn
A2: Feed dog
B1: Feed cat
B7: Feed bird
C3: Mow lawn
C4: Empty trash

```

由于设计上的原因，不能在一个`priority_queue`上从头到尾进行迭代，但是可以用一个`vector`来模拟`priority_queue`的行为，因此允许访问那个`vector`。可以通过观察`priority_queue`的实现来这样做，它使用的函数有`make_heap()`、`push_heap()`以及`pop_heap()`。（这些函数是`priority_queue`的灵魂——事实上，可以说堆就是一个优先队列，`priority_queue`只是对它的一个封装。）结果相当简单，但是读者可能会想，可能还存在一个捷径。因为`priority_queue`使用的容器是`protected`的（并且有标识符，根据标准C++规格说明，该标识符被命名为`c`），所以可以继承一个新类，该新类提供了访问底层实现的途径：

```

//: C07:PriorityQueue4.cpp
// Manipulating the underlying implementation.
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iterator>
#include <queue>
using namespace std;

class PQI : public priority_queue<int> {
public:
    vector<int>& impl() { return c; }
};

int main() {

```



```

PQI pqi;
srand(time(0));
for(int i = 0; i < 100; i++)
    pqi.push(rand() % 25);
copy(pqi.impl().begin(), pqi.impl().end(),
    ostream_iterator<int>(cout, " "));
cout << endl;
while(!pq.empty()) {
    cout << pq.top() << ' ';
    pq.pop();
}
} ///:~

```

然而，如果运行这个程序，就会发现当调用**pop()**时，得到的这个**vector**包含的元素并不是按降序排列，这就是想要从优先队列得到的顺序。似乎如果想要创建一个作为优先队列的**vector**，必须手工完成它。就像下面这样：

```

//: C07:PriorityQueue5.cpp
// Building your own priority queue.
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iterator>
#include <queue>
using namespace std;

template<class T, class Compare>
class PQV : public vector<T> {
    Compare comp;
public:
    PQV(Compare cmp = Compare()) : comp(cmp) {
        make_heap(begin(), end(), comp);
    }
    const T& top() { return this->front(); }
    void push(const T& x) {
        this->push_back(x);
        push_heap(begin(), end(), comp);
    }
    void pop() {
        pop_heap(begin(), end(), comp);
        this->pop_back();
    }
};

int main() {
    PQV< int, less<int> > pq;
    srand(time(0));
    for(int i = 0; i < 100; i++)
        pq.push(rand() % 25);
    copy(pq.begin(), pq.end(),
        ostream_iterator<int>(cout, " "));
    cout << endl;
    while(!pq.empty()) {
        cout << pq.top() << ' ';
        pq.pop();
    }
} ///:~

```

但是这个程序表现得跟前面那个程序一样！读者在**vector**底层中的一个被称为堆（heap）的存储区观察到什么。这个堆数据结构表现为一个优先队列的树的结构（被存储在**vector**的

线性结构中),但是在对其从头到尾进行迭代的时候,并不会得到一个线性的优先队列顺序。你可能认为可以仅调用**sort\_heap()**进行排序,但是那只能起一次作用,然后你将不再拥有一个堆,而只剩一个被排过序的列表。这意味着,要返回将其作为一个堆来使用,用户必须记住首先调用**make\_heap()**。这些都可以被封装到自定义的优先队列中去:

```

//: C07:PriorityQueue6.cpp
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iterator>
#include <queue>
using namespace std;

template<class T, class Compare>
class PQV : public vector<T> {
    Compare comp;
    bool sorted;
    void assureHeap() {
        if(sorted) {
            // Turn it back into a heap:
            make_heap(this->begin(),this->end(), comp);
            sorted = false;
        }
    }
public:
    PQV(Compare cmp = Compare()) : comp(cmp) {
        make_heap(this->begin(),this->end(), comp);
        sorted = false;
    }
    const T& top() {
        assureHeap();
        return this->front();
    }
    void push(const T& x) {
        assureHeap();
        this->push_back(x); // Put it at the end
        // Re-adjust the heap:
        push_heap(this->begin(),this->end(), comp);
    }
    void pop() {
        assureHeap();
        // Move the top element to the last position:
        pop_heap(this->begin(),this->end(), comp);
        this->pop_back();// Remove that element
    }
    void sort() {
        if(!sorted) {
            sort_heap(this->begin(),this->end(), comp);
            reverse(this->begin(),this->end());
            sorted = true;
        }
    }
};

int main() {
    PQV< int, less<int> > pqi;
    srand(time(0));
    for(int i = 0; i < 100; i++) {
        pqi.push(rand() % 25);
        copy(pqi.begin(), pqi.end(),

```



```

        ostream_iterator<int>(cout, " ");
        cout << "\n-----" << endl;
    }
    pqi.sort();
    copy(pqi.begin(), pqi.end(),
        ostream_iterator<int>(cout, " "));
    cout << "\n-----" << endl;
    while(!pq.empty()) {
        cout << pq.top() << ' ';
        pq.pop();
    }
} ///:~

```

如果**sorted**为真，**vector**就不是作为一个堆来进行组织的，而仅仅是个排过序的序列。函数**assureHeap()**保证在对其进行任何堆操作之前使其倒退成为一个堆的形式。**main()**中的第1个**for**循环引入了新的额外特性，它显示一个正在被构建的堆。

在前面的两个程序中采用了“**this->**”前缀的这种表面上并非必要（extraneous）的用法。虽然某些编译器不需要这种用法，但标准C++的定义有这种用法。注意，类**PQV**派生自**vector<T>**，因此继承自**vector<T>**的**begin()**和**end()**都是依赖的名字。<sup>①</sup>在模板的定义中，编译器不能查寻这些来自于依赖的基类的名字（在这种情况下为**vector**），因为对于某个给定的实例，一个明确特化的模板版本可能使用的并不是一个给定的成员。特别的命名需求保证在某些情况下用户不会结束正在调用的一个基类成员，在另外的情况下函数可能来自一个外围空间（比如一个全局的）的函数。编译器无法知道调用的**begin()**是依赖的，因此必须使用“**this->**”限定给它提供一个线索。<sup>②</sup>这就告诉了编译器，这个**begin()**是在**PQV**的范围之内的，因此它就会等待直到**PQV**的一个实例完全地被实例化。如果去掉这个限定前缀，编译器就会对名字**begin**和**end**尝试进行早期查找（在模板定义期间查找，并且会查找失败，因为在这个例子中包含的外围字典空间中并不包括这些名字声明）。然而上面的程序代码中，编译器一直在**pqi**实例化的那一点等待，然后在**vector<int>**中寻找**begin()**和**end()**的正确的特化。

这个解决方案的惟一的缺点就是，用户必须记住在将其作为一个排过序的序列进行查看之前必须先调用**sort()**（一个可以想得到的方法，就是重新定义所有能够产生迭代器的成员函数，以便保证排序的进行）。另一个解决方案就是创建一个非**vector**的优先队列，但是，每当需要时就构建一个使用**vector**的优先队列：

```

//: C07:PriorityQueue7.cpp
// A priority queue that will hand you a vector.
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iterator>
#include <queue>
#include <vector>
using namespace std;

template<class T, class Compare> class PQV {
    vector<T> v;
    Compare comp;
public:

```

① 这意味着它们在以某种方式依赖于一个模板参数。参看第5章中的“名字查找问题”一节。

② 如第5章所述，即任何有效限定，比如**PQV::**，所做的那样。

```

// Don't need to call make_heap(); it's empty:
PQV(Compare cmp = Compare()) : comp(cmp) {}
void push(const T& x) {
    v.push_back(x); // Put it at the end
    // Re-adjust the heap:
    push_heap(v.begin(), v.end(), comp);
}
void pop() {
    // Move the top element to the last position:
    pop_heap(v.begin(), v.end(), comp);
    v.pop_back(); // Remove that element
}
const T& top() { return v.front(); }
bool empty() const { return v.empty(); }
int size() const { return v.size(); }
typedef vector<T> TVec;
TVec getVector() {
    TVec r(v.begin(), v.end());
    // It's already a heap
    sort_heap(r.begin(), r.end(), comp);
    // Put it into priority-queue order:
    reverse(r.begin(), r.end());
    return r;
}
};

int main() {
    PQV<int, less<int> > pqi;
    srand(time(0));
    for(int i = 0; i < 100; i++)
        pqi.push(rand() % 25);
    const vector<int>& v = pqi.getVector();
    copy(v.begin(), v.end(),
        ostream_iterator<int>(cout, " "));
    cout << "\n-----" << endl;
    while(!pq.empty()) {
        cout << pq.top() << ' ';
        pq.pop();
    }
} //:~

```

**PQV**类模板随后采用了与STL的**priority\_queue**相同的形式，但是拥有一个附加的成员函数**getVector()**，该函数创建了一个从**PQV**中（这意味着它已经是一个堆）复制来的新的**vector**。然后它对那些副本进行排序（而**PQV**的**vector**并没有受到影响），并且将序列的顺序逆转，所以在遍历新的**vector**时产生了与从一个优先队列中弹出元素的操作等效的结果。

可以观察到，如果采用在**PriorityQueue4.cpp**中使用的从**priority\_queue**中派生的方法来实现上述技术的话，可以得到更简洁的代码：

```

//: C07:PriorityQueue8.cpp
// A more compact version of PriorityQueue7.cpp.
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iterator>
#include <queue>
using namespace std;

template<class T> class PQV : public priority_queue<T> {
public:
    typedef vector<T> TVec;

```

```

TVec getVector() {
    TVec r(this->c.begin(), this->c.end());
    // c is already a heap
    sort_heap(r.begin(), r.end(), this->comp);
    // Put it into priority-queue order:
    reverse(r.begin(), r.end());
    return r;
}

};

int main() {
    PQV<int> pqi;
    srand(time(0));
    for(int i = 0; i < 100; i++)
        pqi.push(rand() % 25);
    const vector<int>& v = pqi.getVector();
    copy(v.begin(), v.end(),
        ostream_iterator<int>(cout, " "));
    cout << "\n-----" << endl;
    while(!pqi.empty()) {
        cout << pqi.top() << ' ';
        pqi.pop();
    }
} ///:~

```

以上简洁的解决方案，加上它保证用户不会得到一个处在未经排序状态的**vector**，使它变得最简单且最令人期待。惟一潜在的问题就是成员函数**getVector()**采用传值的方式返回**vector<T>**，这可能在参数类型**T**的值比较复杂时会引发某些经常性的开销问题。

## 7.9 持有二进制位

因为C是一种旨在“接近硬件”的语言，但很多人都发现一个令人沮丧的现象，那就是对于数字没有一种固有的二进制的表示方法。当然，有十进制和十六进制（还可以容忍，仅仅因为它们能较容易地在你的头脑中形成一组二进制位），但是八进制呢？哎呀！每当你阅读正在尝试对其进行编程的芯片的说明书时，这些说明书不会使用八进制甚至十六进制来描述芯片的寄存器——他们使用二进制。然而，C不让用**0b0101101**这样的表示方法，很明显，对于接近硬件的语言来说，这才是最好的解决方案。

虽然在C++中仍然没有固有的表示二进制的方法，由于两个类的增加：二进制位集合**bitset**和逻辑向量**vector<bool>**而使得情况有所好转，它们都被设计用来操纵一组开/关值。<sup>①</sup> 这些类型之间主要的不同是：

- 每个**bitset**持有一个固定位数的二进制位（**bit**）。用户在**bitset**的模板参数中建立二进制位的位数。像正常**vector**一样，**vector<bool>**可以动态地扩展为持有任意数目的**bool**值。
- **bitset**模板是为了在操纵二进制位时提高性能的目的而设计，因此并不是一个“正常的”STL容器。因此，它没有迭代器。作为一个模板参数，二进制位的位数目在编译时就已经知道了，并且允许将底层的整型数组存储在运行时的栈上。另一方面，**vector<bool>**容器是**vector**的一个特化，所以有一个普通**vector**的所有操作——该特化只是被设计用来提高**bool**数据的空间使用率。

在**bitset**和**vector<bool>**之间没有琐碎的转换，这意味着它们就是为了完全不同的目的

① Chuck设计并提供了最初的关于**bitset**或**bitstring**、以及**vector<bool>**最早的参考实现，当时，即20世纪90年代早期，他就是C++标准委员会中的一个活跃的成员。



而设计的。此外，它们都不是传统的“STL容器”。**bitset**模板类拥有一个面向二进制位层次的操作接口，绝不与到目前为止本教材中所讨论的STL容器类似。**vector**的特化**vector<bool>**类似于类-STL容器，但与将要在下面讨论的内容也是不同的。

### 7.9.1 **bitset<n>**

**bitset**作为模板接受一个无符号整型模板参数，该参数用来表示二进制位的位数。因此，**bitset<10>**与**bitset<20>**相比就是两种不同的类型，不能在它们两个之间进行比较、赋值等操作。

一个**bitset**以有效的形式提供了最一般的用于二进制位操作的方式。然而，每个**bitset**通过合理地将一组二进制位封装到一个整型数组中来实现（典型的如**unsigned long**，它至少包含32个二进制位）。另外，从一个**bitset**到一个数的惟一转化就是将其转化为一个**unsigned long**（通过函数**to\_ulong()**）。

下面的例子测试了几乎所有**bitset**的功能（这里未介绍那些多余的或不重要的操作）。读者可以在每个打印输出的右边看到对**bitset**输出的描述，因此，可以将这些输出描述与它们原来的值进行比较。如果读者到现在为止还不了解二进制位操作方式的话，运行这个程序将会很有帮助。

```
//: C07:BitSet.cpp {-bor}
// Exercising the bitset class.
#include <bitset>
#include <climits>
#include <cstdlib>
#include <ctime>
#include <cstdint>
#include <iostream>
#include <string>
using namespace std;

const int SZ = 32;
typedef bitset<SZ> BS;

template<int bits> bitset<bits> randBitset() {
    bitset<bits> r(rand());
    for(int i = 0; i < bits/16 - 1; i++) {
        r <<= 16;
        // "OR" together with a new lower 16 bits:
        r |= bitset<bits>(rand());
    }
    return r;
}

int main() {
    srand(time(0));
    cout << "sizeof(bitset<16>) = "
          << sizeof(bitset<16>) << endl;
    cout << "sizeof(bitset<32>) = "
          << sizeof(bitset<32>) << endl;
    cout << "sizeof(bitset<48>) = "
          << sizeof(bitset<48>) << endl;
    cout << "sizeof(bitset<64>) = "
          << sizeof(bitset<64>) << endl;
    cout << "sizeof(bitset<65>) = "
          << sizeof(bitset<65>) << endl;
    BS a(randBitset<SZ>()), b(randBitset<SZ>());
    // Converting from a bitset:
    unsigned long ul = a.to_ulong();
    cout << a << endl;
    // Converting a string to a bitset:
    string cbits("111011010110111");
```



```

cout << "as a string = " << cbits << endl;
cout << BS(cbits) << " [BS(cbits)]" << endl;
cout << BS(cbits, 2) << " [BS(cbits, 2)]" << endl;
cout << BS(cbits, 2, 11) << " [BS(cbits, 2, 11)]" << endl;
cout << a << " [a]" << endl;
cout << b << " [b]" << endl;
// Bitwise AND:
cout << (a & b) << " [a & b]" << endl;
cout << (BS(a) &= b) << " [a &= b]" << endl;
// Bitwise OR:
cout << (a | b) << " [a | b]" << endl;
cout << (BS(a) |= b) << " [a |= b]" << endl;
// Exclusive OR:
cout << (a ^ b) << " [a ^ b]" << endl;
cout << (BS(a) ^= b) << " [a ^= b]" << endl;
cout << a << " [a]" << endl; // For reference
// Logical left shift (fill with zeros):
cout << (BS(a) <= SZ/2) << " [a <= (SZ/2)]" << endl;
cout << (a << SZ/2) << endl;
cout << a << " [a]" << endl; // For reference
// Logical right shift (fill with zeros):
cout << (BS(a) >= SZ/2) << " [a >= (SZ/2)]" << endl;
cout << (a >> SZ/2) << endl;
cout << a << " [a]" << endl; // For reference
cout << BS(a).set() << " [a.set()]" << endl;
for(int i = 0; i < SZ; i++)
    if(!a.test(i)) {
        cout << BS(a).set(i)
            << " [a.set(" << i << ")]" << endl;
        break; // Just do one example of this
    }
cout << BS(a).reset() << " [a.reset()]" << endl;
for(int j = 0; j < SZ; j++)
    if(a.test(j)) {
        cout << BS(a).reset(j)
            << " [a.reset(" << j << ")]" << endl;
        break; // Just do one example of this
    }
cout << BS(a).flip() << " [a.flip()]" << endl;
cout << ~a << " [~a]" << endl;
cout << a << " [a]" << endl; // For reference
cout << BS(a).flip(1) << " [a.flip(1)]" << endl;
BS c;
cout << c << " [c]" << endl;
cout << "c.count() = " << c.count() << endl;
cout << "c.any() = "
    << (c.any() ? "true" : "false") << endl;
cout << "c.none() = "
    << (c.none() ? "true" : "false") << endl;
c[1].flip(); c[2].flip();
cout << c << " [c]" << endl;
cout << "c.count() = " << c.count() << endl;
cout << "c.any() = "
    << (c.any() ? "true" : "false") << endl;
cout << "c.none() = "
    << (c.none() ? "true" : "false") << endl;
// Array indexing operations:
c.reset();
for(size_t k = 0; k < c.size(); k++)
    if(k % 2 == 0)
        c[k].flip();
cout << c << " [c]" << endl;
c.reset();

```



```

// Assignment to bool:
for(size_t ii = 0; ii < c.size(); ii++)
    c[ii] = (rand() % 100) < 25;
cout << c << " [c]" << endl;
// bool test:
if(c[1])
    cout << "c[1] == true";
else
    cout << "c[1] == false" << endl;
} ///:~

```

为产生有趣的随机**bitset**，在程序中创建了函数**randBitset()**。该函数将每16个随机二进制位向左移动，直到**bitset**（其尺寸大小在函数中已经被模板化了）被填满为止，以此来演示**operator<<=**的使用。用**operator|=**将产生的数字和每组新的16位二进制数结合起来。

**main()**函数首先显示了一个**bitset**单元的大小。如果它小于32位，**sizeof**就产生4（4字节 = 32位，其最大实现是一个**long**型的大小。如果它在32到64之间，则需要两个**long**型数，大于64需要3个**long**型，等等。因此，为了最有效地利用空间，所使用的二进制位数量应在适宜个数的固有**long**型数表示的范围中。然而，要注意的是，对该对象不存在额外的开销——就像是在为一个**long**型数进行手工译码一样。

虽然除了**to\_ulong()**之外没有其他的从**bitset**进行数字转换的函数，但是有一个流插入器**stream inserter**，它产生一个包含1和0的**string**，这个字符串可以和实际的**bitset**一样长。

虽然仍然没有用于表示二进制数的基本的格式，但是**bitset**支持最贴近的二进制表示形式：由1和0与在右边的最低有效位（least-significant bit, lsb）一起组成的一个**string**。3个构造函数演示创建一个完整的**string**、在第2个字符开始的**string**以及从第2个字符开始到第11个字符结束的**string**、可以使用**operator<<**从一个**bitset**写到一个输出流**ostream**，它以1和0的方式出现。也可以使用**operator>>**从一个输入流**istream**中读入到**bitset**（在这里没有显示）。

必须注意，**bitset**仅有3个非成员运算符：与（&）、或（|）和异或（^）。其中的每个都创建一个新的**bitset**作为其返回值。在没有创建暂时值的地方，全部选择更有效率的**&=**、**|=**等形式的成员运算符。然而，这些形式改变了**bitset**的值（这个值在上面例子的大多数检测中即**a**）。为避免发生这种情况，通过调用**a**的拷贝构造函数创建一个作为左值的临时对象；这就是为什么**BS(a)**的形式如此。每次测试的结果都显示出来，有时候**a**被重新打印出来从而更容易以它进行参照。

在程序运行的时候，例子的其余部分有自我解释；如果没有，读者可以在自己使用的编译器的文档中或者在本章较早提到的其他文档中查找有关细节。

### 7.9.2 vector<bool>

容器**vector<bool>**是**vector**模板的一个特化。一个标准的**bool**变量至少需要一个字节，但是因为一个**bool**型只有两个状态，所以理想的**vector<bool>**实现是这样的，每一个**bool**值仅需一个二进制位来表示。因为典型的库实现将一组二进制位封装进整型数组之内，所以迭代器必须特殊定义并且不能是一个指向**bool**型的指针。

用于**vector<bool>**的位操纵函数比**bitset**的那些函数受到更多的限制。在**vector**中已有的这些成员函数基础上添加的惟一成员函数就是**flip()**，用于使所有的位取反。它没有**bitset**中的**set()**或**reset()**。当使用**operator[]**时，就会送回一个**vector<bool>::reference**类型的对象，该对象也有一个用于对个别的位取反的成员函数**flip()**。

```

//: C07:VectorOfBool.cpp
// Demonstrate the vector<bool> specialization.
#include <bitset>
#include <cstdint>
#include <iostream>
#include <iterator>
#include <sstream>
#include <vector>
using namespace std;

int main() {
    vector<bool> vb(10, true);
    vector<bool>::iterator it;
    for(it = vb.begin(); it != vb.end(); it++)
        cout << *it;
    cout << endl;
    vb.push_back(false);
    ostream_iterator<bool> out(cout, "");
    copy(vb.begin(), vb.end(), out);
    cout << endl;
    bool ab[] = { true, false, false, true, true,
        true, true, false, false, true };
    // There's a similar constructor:
    vb.assign(ab, ab + sizeof(ab)/sizeof(bool));
    copy(vb.begin(), vb.end(), out);
    cout << endl;
    vb.flip(); // Flip all bits
    copy(vb.begin(), vb.end(), out);
    cout << endl;
    for(size_t i = 0; i < vb.size(); i++)
        vb[i] = 0; // (Equivalent to "false")
    vb[4] = true;
    vb[5] = 1;
    vb[7].flip(); // Invert one bit
    copy(vb.begin(), vb.end(), out);
    cout << endl;
    // Convert to a bitset:
    ostringstream os;
    copy(vb.begin(), vb.end(),
        ostream_iterator<bool>(os, ""));
    bitset<10> bs(os.str());
    cout << "Bitset:" << endl << bs << endl;
} ///:~

```

这个例子中的最后一部分创造了一个**vector<bool>**，通过先将它转换成一个仅包含0和1的**string**，再转换成为一个**bitset**。这里必须在编译时就知道**bitset**的大小。可以看出，这个转换并不是基于常规的那种操作。

某些其他容器保证提供的功能不见了，**vector<bool>**特化给人的感觉是一种“有缺陷的”STL容器。比如，在其他的容器持有如下关系：

```

// Let c be an STL container other than vector<bool>:
T& r = c.front();
T* p = &*c.begin();

```

对于所有其他的容器，**front()**函数产生一个左值（某个对象能获得一个指向它的非常量引用），函数**begin()**必须产生某个对象的解析，并且得到其地址。因为二进制位是不可寻址的，所以上面的两个函数不可能用于处理持有二进制位的容器。**vector<bool>**和**bitset**两者都使用一个代理类（嵌套的**reference**引用类，之前提到过）在必要的时候读取和设置二进制位。

## 7.10 关联式容器

**set**、**map**、**multiset**和**multimap**被称为关联式容器 (associative container)，因为它们将关键字与值关联起来。至少**map**和**multimap**将关键字与值关联在一起，读者可以将一个**set**看成是没有值的**map**，它只有关键字（事实上，它们可以以这样的方式实现），**multiset**和**multimap**之间也有同样的关系。因此，由于结构的相似性，**set**和**multiset**都被归类为关联式容器。

关联式容器最重要的基本操作就是将对象放进容器。并且在**set**的情况下，要查看该对象是否已经在集合中；在**map**的情况下，需要先查看关键字是否已经在**map**中，如果存在，就需要为那个关键字设置关联的值。在这个主题上有很多变化，但是那是基本的概念。下面的例子显示了这些基本操作：

```
//: C07:AssociativeBasics.cpp {-bor}
// Basic operations with sets and maps.
//{L} Noisy
#include <cstdint>
#include <iostream>
#include <iterator>
#include <map>
#include <set>
#include "Noisy.h"
using namespace std;

int main() {
    Noisy na[7];
    // Add elements via constructor:
    set<Noisy> ns(na, na + sizeof na/sizeof(Noisy));
    Noisy n;
    ns.insert(n); // Ordinary insertion
    cout << endl;
    // Check for set membership:
    cout << "ns.count(n)= " << ns.count(n) << endl;
    if(ns.find(n) != ns.end())
        cout << "n(" << n << ") found in ns" << endl;
    // Print elements:
    copy(ns.begin(), ns.end(),
        ostream_iterator<Noisy>(cout, " "));
    cout << endl;
    cout << "\n-----" << endl;
    map<int, Noisy> nm;
    for(int i = 0; i < 10; i++)
        nm[i]; // Automatically makes pairs
    cout << "\n-----" << endl;
    for(size_t j = 0; j < nm.size(); j++)
        cout << "nm[" << j << "] = " << nm[j] << endl;
    cout << "\n-----" << endl;
    nm[10] = n;
    cout << "\n-----" << endl;
    nm.insert(make_pair(47, n));
    cout << "\n-----" << endl;
    cout << "\n nm.count(10)= " << nm.count(10) << endl;
    cout << "nm.count(11)= " << nm.count(11) << endl;
    map<int, Noisy>::iterator it = nm.find(6);
    if(it != nm.end())
        cout << "value:" << (*it).second
            << " found in nm at location 6" << endl;
    for(it = nm.begin(); it != nm.end(); it++)
        cout << (*it).first << ":" << (*it).second << ", ";
}
```



```
    cout << "\n-----" << endl;
} ///:~
```

这里使用两个迭代器来创建`set<Noisy>`对象`ns`，使其进入一个`Noisy`对象的数组之内。但是也有一个默认的构造函数和一个拷贝构造函数，可以传入一个对象以便提供另一种做比较的方案。`set`和`map`两者都有一个成员函数`insert()`用于向其中放入对象，可以用两种方式检查来看看对象是否已经存在于相应的关联式容器中。当给定一个关键字时，成员函数`count()`会告之那个关键字在容器中存在多少次。（在`set`或者`map`中只能是0或者1，但是在`multiset`和`multimap`中则可能多于一个）。成员函数`find()`将会产生一个指向首次出现（在`set`和`map`中则是惟一出现）给定关键字的元素的迭代器，或者如果找不到该关键字，将产生指向超越末尾的迭代器。所有的关联式容器都有`count()`和`find()`成员函数，这是很有意义的。这些关联式容器也都有成员函数`lower_bound()`、`upper_bound()`和`equal_range()`，它们仅仅对`multiset`和`multimap`有意义，正如读者所见。（但是不要试图去搞清楚它们对`set`和`map`到底有什么用，因为它们被设计用来处理某个范围的重复关键字，这在`set`和`map`容器中是不允许的。）

设计一个`operator[]`总是多少有点进退两难。因为它被有意地作为一个数组索引操作来对待，人们在使用前一般不会想到对其进行测试。但是，假如决定将索引设置为超出数组范围以外的位置时会发生什么事情？一种选择是抛出一个异常，但是对于一个`map`，“在数组范围以外的位置进行索引”意味着希望在那个位置创建一个新条目，这就是STL `map`的处理方式。在创建`map<int, Noisy> nm`之后的第1个`for`循环使用`operator[]`来“查找”对象，但实际上这是在创建新的`Noisy`对象！如果使用`operator[]`查寻一个值而它又不存在的话，这个`map`就会创建一个新的关键字-值对（为这个值使用默认的构造函数）。这意味着，如果实际上仅想要查寻某个对象并不想创建一个新条目，就必须使用成员函数`count()`（看这个对象是否在那里）或者`find()`（得到指向它的迭代器）。

与`for`循环一起使用运算符`operator[]`来打印容器中的值会有许多问题。首先，它需要整数关键字（在这里恰好是这样）。其次且更糟的是，如果所有的关键字都不是连续的，那么将会完成从0到整个容器的大小全部都进行计算，如果某些点没有关键字-值对存在，系统将会自动创建它们并会错过一些较高值的关键字。最后，如果观察`for`循环的输出，将会看到其工作非常繁忙。起先读者会相当迷惑，一个简单的查寻怎么会出现如此多的构造与析构呢？只有当看到`map`模板中关于`operator[]`的代码时答案才清楚为什么，这段代码如下所示：

```
mapped_type& operator[] (const key_type& k) {
    value_type tmp(k, T());
    return (*((insert(tmp)).first)).second;
}
```

函数`map::insert()`接受一个关键字-值对，如果在映像中已有与给定的关键字在一起的条目，就什么也不做——否则它为该关键字插入一个条目。在两者之中任一情况下，它返回一个新的关键字-值对，该关键字-值对的第1个元素持有指向被插入对的迭代器，如果发生了插入操作，该对的第2个元素持有值为真。成员`first`和`second`分别给出了关键字和值，因为`map::value_type`实际上只是一个为`std::pair`进行类型定义的`typedef`：

```
typedef pair<const Key, T> value_type;
```

读者已经在前面看到了`std::pair`模板。它是两个独立类型值的简单持有者，就像在其定义中所看到的那样：

```

template<class T1, class T2> struct pair {
    typedef T1 first_type;
    typedef T2 second_type;
    T1 first;
    T2 second;
    pair();
    pair(const T1& x, const T2& y) : first(x), second(y) {}
    // Templated copy-constructor:
    template<class U, class V> pair(const pair<U, V> &p);
};

```

**pair**模板类非常有用，特别是想要一个函数返回两个对象的时候（因为一个**return**语句只能返回一个对象）。为了创建一个关键字-值对**pair**，甚至还有一个快捷的称为**make\_pair()**的函数，它用在**AssociativeBasics.cpp**中。

追溯上面执行的各个步骤，**map::value\_type**是**map**的一个关键字-值对**pair**——实际上，它是**map**的一个条目。但是要注意，**pair**由值封装它的对象，这意味着将对象装入**pair**之内，拷贝构造是必须的。因此，在**map::operator[]**的**tmp**创建过程中，对于每个**pair**中的对象将包括至少一个拷贝构造函数调用和一个析构造函数调用。在这里，可以很容易地完成这些操作，因为关键字是**int**型的。但是，如果想要看看根据**map::operator[]**的活动方式到底能产生什么样的结果，请运行下面这个程序：

```

//: C07:NoisyMap.cpp
// Mapping Noisy to Noisy.
//{L} Noisy
#include <map>
#include "Noisy.h"
using namespace std;

int main() {
    map<Noisy, Noisy> mnn;
    Noisy n1, n2;
    cout << "\n-----" << endl;
    mnn[n1] = n2;
    cout << "\n-----" << endl;
    cout << mnn[n1] << endl;
    cout << "\n-----" << endl;
} ///:~

```

读者将会看到，插入和查寻两者都会产生很多额外的对象，这是因为**tmp**对象的创建。如果回过来看**map::operator[]**，就会看到第2行调用了**insert()**，并向其传递**tmp**——即**operator[]**每次都进行了插入操作。函数**insert()**的返回值是一种不同的**pair**类型，其**first**是一个指向刚刚插入的关键字-值对的迭代器，而**second**则是一个表示在该处是否发生了插入操作的**bool**值。可以看到，**operator[]**抓取了**first**（迭代器），对其进行解析以产生**pair**，然后返回**second**，即该位置上的值。

因此，从上面的描述看来，**map**具有“如果在那里没有条目的话就创建一个”的奇妙行为，但是从下面（具体操作）来看，就是在使用**map::operator[]**时总是得到很多额外的对象创建和析构操作。幸运的是，**AssociativeBasics.cpp**也演示了如何减少插入和删除操作的开销。如果不需要它，尽量避免使用**operator[]**。成员函数**insert()**比**operator[]**稍微更有效些。对于一个**set**，仅仅持有一个对象，而对于**map**来说，持有的是关键字-值对；所以**insert()**需要一个**pair**作为其参数。这里就是**make\_pair()**派得上用场的地方，就像在程序中所能看到的那样。

为了在一个**map**中查寻对象，可以使用**count()**来查看这个关键字是否在**map**中，或者

可以用**find()**产生一个直接指向关键字-值对的迭代器。再次强调,因为**map**包含**pair**,这就是在解析它的时候为什么会产生迭代器的原因,所以选择**first**和**second**作为其参数。在运行**AssociativeBasics.cpp**的时候,读者将会注意到,使用迭代器的方法不会产生额外的对象的构造和析构操作。然而,就易编写或者易阅读的面向对象编码要求而言,这是不可取的。

### 7.10.1 用于关联式容器的发生器和填充器

在使用**<algorithm>**中的**fill()**、**fill\_n()**、**generate()**和**generate\_n()**函数模板向序列容器(**vector**、**list**和**deque**)中填充数据时,已经看到了它们是多么的有用。然而,它们的实现都使用**operator=**赋值的方式将值放进序列容器,而向关联式容器中添加对象的方式是使用它们各自的成员函数**insert()**。因此,在尝试与关联式容器一起使用“填充(fill)”和“产生(generate)”函数的时候,默认的“赋值”行为将会产生问题。

一个解决方案就是复制“填充”和“产生”函数,创建新的一种能用于关联式容器的函数。结果是,只有**fill\_n()**和**generate\_n()**函数能被复制(**fill()**和**generate()**复制序列,这对于关联式容器来说没有什么意义),但是这个工作是相当简单的,因为可以利用头文件**<algorithm>**作为工作的根据:

```

//: C07:assocGen.h
// The fill_n() and generate_n() equivalents
// for associative containers.
#ifndef ASSOCGEN_H
#define ASSOCGEN_H

template<class Assoc, class Count, class T>
void assocFill_n(Assoc& a, Count n, const T& val) {
    while(n-- > 0)
        a.insert(val);
}

template<class Assoc, class Count, class Gen>
void assocGen_n(Assoc& a, Count n, Gen g) {
    while(n-- > 0)
        a.insert(g());
}
#endif // ASSOCGEN_H ///:~

```

读者可以看到,没有使用迭代器,容器类自身被传递了(当然,通过使用引用)。

这段代码演示了两条有价值的经验教训。第1条就是,如果有什么需要的工作某个算法不能做,可以复制与其最接近的算法,并且修改它以满足需要。在STL头文件中有很多手到擒来例子,从这一点来说,大多数工作实际上已经完成了。

第2条经验教训进一步指出:如果观察的时间足够长,就会发现在STL中有一种方法来做这个工作,而不必再发明任何新的东西。当前的问题可以用**insert\_iterator**(调用**inserter()**而产生)来解决,它调用**insert()**而非**operator=**以便在容器中放入对象。这不是仅仅对**front\_insert\_iterator**或者**back\_insert\_iterator**的变更,因为那些迭代器使用各自的**push\_front()**和**push\_back()**。每个插入迭代器都因为其用于插入操作的成员函数各具的优点而不尽相同,**insert()**正是我们所需要的一个函数。这里有一个演示显示进行填充和产生**map**和**set**两个容器的例子。(它也可以用于**multiset**和**multimap**。)首先,创建一些模板化的发生器。(这似乎像是有点过分,但在需要它们的时候,用户绝不会知道。为此,它们被放置在一个头文件中。)



```

//: C07:SimpleGenerators.h
// Generic generators, including one that creates pairs.
#include <iostream>
#include <utility>

// A generator that increments its value:
template<typename T> class IncrGen {
    T i;
public:
    IncrGen(T ii) : i(ii) {}
    T operator()() { return i++; }
};

// A generator that produces an STL pair<>:
template<typename T1, typename T2> class PairGen {
    T1 i;
    T2 j;
public:
    PairGen(T1 ii, T2 jj) : i(ii), j(jj) {}
    std::pair<T1,T2> operator()() {
        return std::pair<T1,T2>(i++, j++);
    }
};

namespace std {
// A generic global operator<< for printing any STL pair<>:
template<typename F, typename S> ostream&
operator<<(ostream& os, const pair<F,S>& p) {
    return os << p.first << "\t" << p.second << endl;
}
} //::~~

```

两个发生器都希望**T**可以进行增1操作，无论用什么来进行初始化，它们都仅使用**operator++**来产生新的值。**PairGen**创建一个STL **pair**对象作为其返回值，这也就是为什么可以使用**insert()**向一个**map**或者**multimap**中放入对象的原因。

最后的函数是个一般用于输出流**ostream**的操作符**operator<<**，假定**pair**的每一个元素都支持流操作符**operator<<**，因此任何**pair**都能被打印。（这是在第5章中讨论过的名字空间**std**中奇怪的名字查寻的推论，在本章**Thesaurus.cpp**之后将再一次解释。）如下所示，这允许用**copy()**来输出**map**：

```

//: C07:AssocInserter.cpp
// Using an insert_iterator so fill_n() and generate_n()
// can be used with associative containers.
#include <iterator>
#include <iostream>
#include <algorithm>
#include <set>
#include <map>
#include "SimpleGenerators.h"
using namespace std;

int main() {
    set<int> s;
    fill_n(inserter(s, s.begin()), 10, 47);
    generate_n(inserter(s, s.begin()), 10,
        IncrGen<int>(12));
    copy(s.begin(), s.end(),
        ostream_iterator<int>(cout, "\n"));
    map<int, int> m;
    fill_n(inserter(m, m.begin()), 10, make_pair(90,120));
}

```



```

generate_n(inserter(m, m.begin()), 10,
    PairGen<int, int>(3, 9));
copy(m.begin(), m.end(),
    ostream_iterator<pair<int, int> >(cout, "\n"));
} ///:~

```

传递给**inserter**的第2个参数是一个迭代器，它是最佳化的，暗示可以帮助较快地进行插入（而不总是从底层树形结构的根开始进行搜索）。因为**insert\_iterator**可以用于很多不同类型的容器，对于非-关联式容器来说，它还有更多的意义——它是必需的。

注意**ostream\_iterator**是如何被创建来输出一个**pair**的。如果未创建**operator<<**，它不会起什么作用。因为它是一个模板，它将自动地为**pair<int,int>**进行实例化。

### 7.10.2 不可思议的映像

一个普通的数组使用一个整数值来对连续排列的某种类型的元素集进行索引。**map**是一个关联式数组（associative array），这意味着，按照类数组的方式将一个对象与另一个对象关联到一起。而不是像处理普通数组的方式一样使用一个数字来选择某个数组元素，在这里利用一个对象来进行查寻！下面的例子对一个文本文件中的单词进行计数，因此索引是一个代表单词的**string**对象，被查寻的值就是保存字串（单词）总数的对象。

在一个类似于**vector**或**list**的单项容器中，仅保存着一样东西。但是在一个**map**中，将会得到两样东西：关键字（key）（用它来进行查寻，就像在**mapname[key]**中）以及作为对关键字进行查寻得到的结果值。如果只希望遍历整个**map**并列出每一个关键字-值对的话，可以使用一个迭代器，它在解析时产生一个包含了关键字及其值的**pair**对象。可以通过选择**first**和**second**访问**pair**对象中的成员。

这种将两项一起进行打包的相同思想也用于将元素插入**map**的操作，但是包含了关键字及值的**pair**是作为**map**实例化的一部分来进行创建的，该**pair**称为**value\_type**。所以插入新元素操作的一个选择就是创建一个**value\_type**对象，以适当的对象装载它，然后为**map**调用**insert()**成员函数进行插入操作。下面的例子使用了上述的**map**的特性：如果尝试向**operator[]**传递一个关键字来查找某个对象，当那个对象不存在时，**operator[]**将会自动使用值对象的默认构造函数插入一个新的关键字-值对。以这种思想为基础，现在考虑一个单词计数程序的实现：

```

//: C07:WordCount.cpp
// Count occurrences of words using a map.
#include <iostream>
#include <fstream>
#include <map>
#include <string>
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
    typedef map<string, int> WordMap;
    typedef WordMap::iterator WMIter;
    const char* fname = "WordCount.cpp";
    if(argc > 1) fname = argv[1];
    ifstream in(fname);
    assure(in, fname);
    WordMap wordmap;
    string word;
    while(in >> word)
        wordmap[word]++;
    for(WMIter w = wordmap.begin(); w != wordmap.end(); w++)

```



```
    cout << w->first << ": " << w->second << endl;
} ///:~
```

这个例子显示了零初始化 (zero-initialization) 的能力。考虑程序代码中的这行:

```
wordmap[word]++;
```

这个将 **int** 与 **word** 关联在一起的表达式进行增1操作。如果 **map** 映像中没有这样的一个单词, 则作为该单词的关键字-值对就会自动地插入到 **map** 映像中, 并调用返回值为0的伪构造函数 **int()** 并将其值初始化为0。

打印整个列表需要一个能够遍历该列表的迭代器。(这里不存在用于 **map** 的快捷方式的 **copy()**, 除非需要再为 **map** 中的 **pair** 编写一个 **operator<<**。)如前所述, 解析该迭代器会产生一个 **pair** 对象, 其中 **first** 成员为关键字, **second** 成员为其值。

如果希望找到为某个特定单词的计数, 可以使用数组的索引操作符, 如下所示:

```
cout << "the: " << wordmap["the"] << endl;
```

可以看到, **map** 的主要优点之一就是其清楚的语法; 一个关联式数组对于读者来说是直观的。(然而要注意的是, 如果单词 “the” 已不在 **wordmap** 中, 一个新的条目就会被创建!)

### 7.10.3 多重映像和重复的关键字

多重映像 **multimap** 是一个能包含重复的关键字的 **map**。起初这可能似乎是一个奇怪的想法, 但令人惊讶的这种情况却经常发生。比如电话号码簿, 同一个名字可以有很多个条目。

假定读者正在监视野生动植物, 需要跟踪每一种有斑点的动物出现的时间和地点。因此, 你就可能看到很多同一种类的动物, 它们都在不同的时间和不同的地点出现。因此, 如果将动物的类型作为关键字, 就需要一个 **multimap**。如下所示:

```
//: C07:WildLifeMonitor.cpp
#include <algorithm>
#include <cstdlib>
#include <cstdint>
#include <ctime>
#include <iostream>
#include <iterator>
#include <map>
#include <sstream>
#include <string>
#include <vector>
using namespace std;

class DataPoint {
    int x, y; // Location coordinates
    time_t time; // Time of Sighting
public:
    DataPoint() : x(0), y(0), time(0) {}
    DataPoint(int xx, int yy, time_t tm) :
        x(xx), y(yy), time(tm) {}
    // Synthesized operator=, copy-constructor OK
    int getX() const { return x; }
    int getY() const { return y; }
    const time_t* getTime() const { return &time; }
};

string animal[] = {
    "chipmunk", "beaver", "marmot", "weasel",
    "squirrel", "ptarmigan", "bear", "eagle",
    "hawk", "vole", "deer", "otter", "hummingbird",
};
const int ASZ = sizeof animal/sizeof *animal;
```



```

vector<string> animals(animal, animal + ASZ);

// All the information is contained in a
// "Sighting," which can be sent to an ostream:
typedef pair<string, DataPoint> Sighting;

ostream&
operator<<(ostream& os, const Sighting& s) {
    return os << s.first << " sighted at x= "
        << s.second.getX() << ", y= " << s.second.getY()
        << ", time = " << ctime(s.second.getTime());
}

// A generator for Sightings:
class SightingGen {
    vector<string>& animals;
    enum { D = 100 };
public:
    SightingGen(vector<string>& an) : animals(an) {}
    Sighting operator()() {
        Sighting result;
        int select = rand() % animals.size();
        result.first = animals[select];
        result.second = DataPoint(
            rand() % D, rand() % D, time(0));
        return result;
    }
};

// Display a menu of animals, allow the user to
// select one, return the index value:
int menu() {
    cout << "select an animal or 'q' to quit: ";
    for(size_t i = 0; i < animals.size(); i++)
        cout << '[' << i << ']' << animals[i] << ' ';
    cout << endl;
    string reply;
    cin >> reply;
    if(reply.at(0) == 'q') return 0;
    istringstream r(reply);
    int i;
    r >> i; // Converts to int
    i %= animals.size();
    return i;
}

int main() {
    typedef multimap<string, DataPoint> DataMap;
    typedef DataMap::iterator DMIter;
    DataMap sightings;
    srand(time(0)); // Randomize
    generate_n(inserter(sightings, sightings.begin()),
        50, SightingGen(animals));
    // Print everything:
    copy(sightings.begin(), sightings.end(),
        ostream_iterator<Sighting>(cout, ""));
    // Print sightings for selected animal:
    for(int count = 1; count < 10; count++) {
        // Use menu to get selection:
        // int i = menu();
        // Generate randomly (for automated testing):
        int i = rand() % animals.size();
        // Iterators in "range" denote begin, one

```



```

        // past end of matching range:
        pair<DMIter, DMIter> range =
            sightings.equal_range(animals[i]);
        copy(range.first, range.second,
            ostream_iterator<Sighting>(cout, " "));
    }
} ///:-

```

将观察到的所有数据都封装到一个**DataPoint**类中，该类非常简单足以使用综合赋值和拷贝构造函数来对其进行操作。它用标准C库的时间函数来记录观察的时间。

在**string**数组**animal**中，要注意的是在初始化期间，**char\***构造函数将会自动地调用，这就使得对**string**数组进行初始化变得相当方便。因为在一个**vector**中较易使用动物的名字，所以在计算好数组的长度后，使用构造函数**vector(iterator, iterator)**来初始化一个**vector<string>**。

用于构建**Sighting**的关键字-值对是表示动物类型名称的**string**。**DataPoint**表示观察到该动物的时间和地点。标准的**pair**模板将这两个类型关联起来，并且使用类型定义产生**Sighting**类型。然后为**Sighting**创建一个**ostream operator<<**；这将允许对一个存储了**Sighting**的**map**或**multimap**进行迭代并显示它。

**SightingGen**产生在随机数据点随机观察到的数据用于测试。它有一个普通的**operator()**，这对于函数对象来说是必需的，但是它还有一个构造函数，用于获得和存储一个引用到**vector<string>**，这就是前面提到的存储动物名称的地方。

**DataMap**是一个包含了**string-DataPoint**对的**multimap**，这意味着它存储**Sighting**对象。用**generate\_n()**产生的50个**Sighting**对象来填充该**DataMap**，并且显示它们。（注意，因为存在一个接受**Sighting**的**operator<<**，所以可以创建一个输出流迭代器**ostream\_iterator**。）此时就可以请用户选择他们想要查看所有观察记录中的哪一种动物的情况。如果键入**q**程序就会退出，但是如果选择一个动物的编号，就会调用**equal\_range()**成员函数。这将会返回一个指向匹配对**pair**集的起始元素的迭代器（**DMIter**）和一个指向该匹配对**pair**集的超越末尾的迭代器。因为从一个函数中只能返回一个对象，因此**equal\_range()**使用了**pair**。因为**range**对拥有匹配集的起始和终止迭代器，所以这些迭代器可以在**copy()**函数中用来打印对某种特定类型动物的所有观察记录。

#### 7.10.4 多重集合

读者已经知道**set**仅允许插入每个值的惟一个对象。而**multiset**看起来则比较古怪，因为它允许插入每个值的多个对象。这似乎违反了“集合”的完整的思想，读者可能会问，“它在这个集合中吗？”如果集合中存在着多个“它”，将意味着什么呢？

想一想就会明白，如果这些重复的对象确实完全相同，在一个集合中有多个相同值的对象意义并不大（对那些出现的对象进行计数的情况可能是一个例外，但是，就像在本章较早时看到的，这个问题可以使用另一种更优雅的方法来处理）。因此，每个重复的对象都应该有什么地方“不同于”其他的重复对象——最有可能是在比较期间那些未被用作关键字计算的不同的状态信息。也就是说，通过比较操作这些对象看起来相同，但是它们却包括一些不同的内部状态。

像任何STL容器都必须对其元素进行排序一样，**multiset**模板在默认情况下使用**less**函数对象来决定元素的顺序。它使用了被包含类的比较运算符**operator<**，但可以允许用户用自己的比较函数来代替它。

考虑一个简单的包含一个用于进行比较的元素与另一个不用于进行比较的元素的类：

```

//: C07:MultiSet1.cpp
// Demonstration of multiset behavior.
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iterator>
#include <set>
using namespace std;

class X {
    char c; // Used in comparison
    int i; // Not used in comparison
    // Don't need default constructor and operator=
    X();
    X& operator=(const X&);
    // Usually need a copy-constructor (but the
    // synthesized version works here)
public:
    X(char cc, int ii) : c(cc), i(ii) {}
    // Notice no operator== is required
    friend bool operator<(const X& x, const X& y) {
        return x.c < y.c;
    }
    friend ostream& operator<<(ostream& os, X x) {
        return os << x.c << ":" << x.i;
    }
};

class Xgen {
    static int i;
    // Number of characters to select from:
    enum { SPAN = 6 };
public:
    X operator()() {
        char c = 'A' + rand() % SPAN;
        return X(c, i++);
    }
};

int Xgen::i = 0;

typedef multiset<X> Xmset;
typedef Xmset::const_iterator Xmit;

int main() {
    Xmset mset;
    // Fill it with X's:
    srand(time(0)); // Randomize
    generate_n(inserter(mset, mset.begin()), 25, Xgen());
    // Initialize a regular set from mset:
    set<X> unique(mset.begin(), mset.end());
    copy(unique.begin(), unique.end(),
        ostream_iterator<X>(cout, " "));
    cout << "\n---" << endl;
    // Iterate over the unique values:
    for(set<X>::iterator i = unique.begin();
        i != unique.end(); i++) {
        pair<Xmit, Xmit> p = mset.equal_range(*i);
        copy(p.first, p.second, ostream_iterator<X>(cout, " "));
        cout << endl;
    }
} ///:~

```

在**X**中，所有的比较都产生**char c**。因为在这个例子中使用了默认的**less**比较对象，比较由**operator<**进行，这就是**multiset**所必需的全部工作。类**Xgen**随机产生**X**对象，但是用于比较的值被限制在'A'到'E'之间的范围内。在**main()**函数中，创建一个**multiset<X>**并用**Xgen**向其中填入25个**X**对象，这就保证了那里存在重复的关键字。所以为了了解存在哪些惟一的键值，根据**multiset**创建了一个常规的**set<X>**（使用**iterator**、**iterator**构造函数）。这些值被显示出来，然后对在**multiset**中的每个关键字值都产生**equal\_range()**（**equal\_range()**在这里和在**multimap**中有着相同的意义：所有的元素都与进行匹配的关键字相匹配）。然后打印每个匹配的关键字集。

作为第2个例子，用**multiset**创建的一个（可能）版本更优雅的**WordCount.cpp**：

```

//: C07:MultiSetWordCount.cpp
// Count occurrences of words using a multiset.
#include <fstream>
#include <iostream>
#include <iterator>
#include <set>
#include <string>
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
    const char* fname = "MultiSetWordCount.cpp";
    if(argc > 1) fname = argv[1];
    ifstream in(fname);
    assure(in, fname);
    multiset<string> wordmset;
    string word;
    while(in >> word)
        wordmset.insert(word);
    typedef multiset<string>::iterator MSit;
    MSit it = wordmset.begin();
    while(it != wordmset.end()) {
        pair<MSit, MSit> p = wordmset.equal_range(*it);
        int count = distance(p.first, p.second);
        cout << *it << ": " << count << endl;
        it = p.second; // Move to the next word
    }
}
//:~

```

**main()**函数中的设置与**WordCount.cpp**中完全相同，然后每个单词都只是被插入到**multiset<string>**中。一个迭代器被创建出来并且初始化为指向**multiset**的起始处；解析该迭代器就可以产生它所指向的当前单词。成员函数**equal\_range()**（并非通用算法）产生当前选中的单词的起始和终止迭代器，算法**distance()**（在**<iterator>**中定义的）对该范围内的元素进行计数。然后，迭代器**it**向前移动到范围终止之处，并令其指向下一个单词。如果读者现在还不熟悉**multiset**的话，那么这里的代码就可能显得太复杂。但它的紧凑性和没有所需的诸如**Count**这样的支持类却具有很强的吸引力。

最后，这个容器到底是个真实地“集合”，或者还是应当使用别的名字来命名它呢？另一种选择是，可以将其命名为在某些容器库中定义的一般称为“袋子（bag）”的容器，因为一个袋子可以不加区别地保存任何东西——包括重复的对象。这样的命名比较接近实际情况，可是由于袋子没有对元素按怎样的顺序排列给出规范，所以它也不是完全适合。**multiset**（它要求所有重复的元素必须相互毗邻地存放）比起**set**的概念来其限制甚至更加严格。一个**set**实现可能使用散列函数（hashing function）来排列其元素，这样它将不会按排序的顺序存放这些元

素。另外，如果想要不受限制地（即没有任何指定标准）存放一个对象串，可以使用**vector**，**deque**或者**list**。

## 7.11 将STL容器联合使用

在使用一个同义语词汇编（thesaurus）时，读者可能想知道所有与某个特定单词相似的所有单词。在查寻一个单词的时候，读者希望结果由一个单词表给出。这里，那些“多重”容器（**multiset**或者**multimap**）都不适合。解决方案是将容器联合在一起使用，该方法用STL很容易实现。在这里，我们需要一个工具，其结果是形成一个功能强大的通用的概念，那就是能使字符串与一个**vector**关联成为**map**：

```
//: C07:Thesaurus.cpp
// A map of vectors.
#include <map>
#include <vector>
#include <string>
#include <iostream>
#include <iterator>
#include <algorithm>
#include <ctime>
#include <cstdlib>
using namespace std;

typedef map<string, vector<string> > Thesaurus;
typedef pair<string, vector<string> > TEntry;
typedef Thesaurus::iterator TIter;

// Name lookup work-around:
namespace std {
ostream& operator<<(ostream& os,const TEntry& t) {
    os << t.first << ": ";
    copy(t.second.begin(), t.second.end(),
        ostream_iterator<string>(os, " "));
    return os;
}
}

// A generator for thesaurus test entries:
class ThesaurusGen {
    static const string letters;
    static int count;
public:
    int maxSize() { return letters.size(); }
    TEntry operator()() {
        TEntry result;
        if(count >= maxSize()) count = 0;
        result.first = letters[count++];
        int entries = (rand() % 5) + 2;
        for(int i = 0; i < entries; i++) {
            int choice = rand() % maxSize();
            char cbuf[2] = { 0 };
            cbuf[0] = letters[choice];
            result.second.push_back(cbuf);
        }
        return result;
    }
};

int ThesaurusGen::count = 0;
```





```

const string ThesaurusGen::letters("ABCDEFGHijkl"
    "MNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz");

// Ask for a "word" to look up:
string menu(Thesaurus& thesaurus) {
    while(true) {
        cout << "Select a \"word\", 0 to quit: ";
        for(TIter it = thesaurus.begin();
            it != thesaurus.end(); it++)
            cout << (*it).first << ' ';
        cout << endl;
        string reply;
        cin >> reply;
        if(reply.at(0) == '0') exit(0); // Quit
        if(thesaurus.find(reply) == thesaurus.end())
            continue; // Not in list, try again
        return reply;
    }
}

int main() {
    srand(time(0)); // Seed the random number generator
    Thesaurus thesaurus;
    // Fill with 10 entries:
    generate_n(inserter(thesaurus, thesaurus.begin()),
        10, ThesaurusGen());
    // Print everything:
    copy(thesaurus.begin(), thesaurus.end(),
        ostream_iterator<TEntry>(cout, "\n"));
    // Create a list of the keys:
    string keys[10];
    int i = 0;
    for(TIter it = thesaurus.begin();
        it != thesaurus.end(); it++)
        keys[i++] = (*it).first;
    for(int count = 0; count < 10; count++) {
        // Enter from the console:
        // string reply = menu(thesaurus);
        // Generate randomly
        string reply = keys[rand() % 10];
        vector<string>& v = thesaurus[reply];
        copy(v.begin(), v.end(),
            ostream_iterator<string>(cout, " "));
        cout << endl;
    }
} //::~~

```

**Thesaurus**将一个**string**（即单词）映射到一个**vector<string>**（同义词）。**TEntry**是**Thesaurus**中的一个条目。通过为**TEntry**创建一个输出流操作符**ostream operator<<**，可以很容易地打印来自**Thesaurus**中的这一条目（而整个**Thesaurus**可以容易地用**copy()**进行打印）。注意，流插入符所处的非常奇怪的位置：它被放置在**std**名字空间中！<sup>①</sup>上面的这个**operator<<()**函数将在**main()**中的第1个**copy()**调用中被**ostream\_iterator**使用。在编译器实例化时，所需要的**ostream\_iterator**特化根据参数关联查找（argument-dependent lookup, ADL）规则，它只查看**std**，因为函数**copy()**所有的参数都在那儿声明。如果在全局名字空间中声明插入符（将其范围限定为迁移名字空间块），它就不会被发现。将

① 从技术上讲，用户向标准名字空间中添加东西是不合法的，但这是防止出现隐藏的名字查找问题的最简单的方法，并且被使用的所有编译器支持。

其放入**std**中，就可以通过ADL找到它。

**ThesaurusGen**创建“单词”（它们仅是单个字母）以及这些单词的“同义词”（这是另外一些随机选择的单个字母）以用作同义语词汇编的条目。它随机挑选制造同义词条目的个数，但必须至少为两个。所有被选择的字母都被编进一个静态字符串**static string**索引中，该**static string**是**ThesaurusGen**的一部分。

在**main()**函数中，创建一个**Thesaurus**，并填入10个条目，并且调用**copy()**算法将它们打印出来。函数**menu()**要求用户通过键入代表单词的字母来选择一个“单词”来进行查询。用成员函数**find()**来查找以确定该条目是否在**map**中。（记住不要使用**operator[]**，它将会在未找到匹配条目的情况下自动创建一个新的条目！）如果存在，就用**operator[]**取出**vector<string>**进行显示。对于**reply**字符串的选择是随机产生的，允许进行自动测试。

模板的使用使得表达功能强大的概念变得很容易，甚至可以更进一步地扩展这个概念创建一个**vector**的**map**，而**vector**又包含有**map**等等。由于这个原因，可以用这种方法联合任何STL容器。

## 7.12 清除容器的指针

在**Stlshape.cpp**中，容器中的那些指针自己不会自动清除。有方便的方法能很容易地做这些事情，不必每一次都为此编写专用代码。这里有个能够清除任何序列容器中指针的函数模板。注意，它被放置在本教材的根目录下面以方便使用：

```
//: :purge.h
// Delete pointers in an STL sequence container.
#ifndef PURGE_H
#define PURGE_H
#include <algorithm>

template<class Seq> void purge(Seq& c) {
    typename Seq::iterator i;
    for(i = c.begin(); i != c.end(); ++i) {
        delete *i;
        *i = 0;
    }
}

// Iterator version:
template<class InpIt> void purge(InpIt begin, InpIt end) {
    while(begin != end) {
        delete *begin;
        *begin = 0;
        ++begin;
    }
}
#endif // PURGE_H ///:~
```

在**purge()**的第1版中，要注意关键字**typename**是绝对必需的。该关键字正是设计用来解决问题的：**Seq**是一个模板参数，而**iterator**则是嵌套在该模板中的某种东西。那么**Seq::iterator**做什么用呢？关键字**typename**说明，它提到的是个类型，而不是其他什么东西。

虽然**purge()**的容器版本必须与一个STL风格的容器一起工作，但**purge()**的迭代器版本的工作区域则涵盖了所有范围，包括数组。

这里有一个重写了的**Stlshape.cpp**，修改并使用了**purge()**函数：

```

//: C07:Stlshape2.cpp
// Stlshape.cpp with the purge() function.
#include <iostream>
#include <vector>
#include "../purge.h"
using namespace std;

class Shape {
public:
    virtual void draw() = 0;
    virtual ~Shape() {};
};

class Circle : public Shape {
public:
    void draw() { cout << "Circle::draw" << endl; }
    ~Circle() { cout << "~Circle" << endl; }
};

class Triangle : public Shape {
public:
    void draw() { cout << "Triangle::draw" << endl; }
    ~Triangle() { cout << "~Triangle" << endl; }
};

class Square : public Shape {
public:
    void draw() { cout << "Square::draw" << endl; }
    ~Square() { cout << "~Square" << endl; }
};

int main() {
    typedef std::vector<Shape*> Container;
    typedef Container::iterator Iter;
    Container shapes;
    shapes.push_back(new Circle);
    shapes.push_back(new Square);
    shapes.push_back(new Triangle);
    for(Iter i = shapes.begin(); i != shapes.end(); i++)
        (*i)->draw();
    purge(shapes);
} ///:~

```

在使用**purge()**时，要仔细考虑该函数的所有权问题。如果在多个容器中持有同一个对象的指针，要确信不对其进行两次删除操作。不希望在第2个容器结束对该对象的使用之前就在第1个容器中将其销毁。对一个容器进行两次清除操作**purge()**不会产生问题，因为**purge()**在删除一个指针后将其值置为零，对一个零指针调用删除操作**delete**是一个安全的操作。

### 7.13 创建自己的容器

有了STL作基础，用户就可以创建自己的容器了。假定读者根据提供的迭代器进行模仿，用户自己创建的新容器将会表现得就好像一个内置的STL容器。

考虑某个“环形”数据结构，它是一个循环的序列容器。如果到达了环的末尾端点，即此时它刚好是绕回到起始端点（末尾端点和起始端点是同一个点）。这可以在熟练掌握**list**的基础上实现，如下所示：

```

//: C07:Ring.cpp
// Making a "ring" data structure from the STL.
#include <iostream>
#include <iterator>
#include <list>
#include <string>
using namespace std;

template<class T> class Ring {
    list<T> lst;
public:
    // Declaration necessary so the following
    // 'friend' statement sees this 'iterator'
    // instead of std::iterator:
    class iterator;
    friend class iterator;
    class iterator : public std::iterator<
        std::bidirectional_iterator_tag, T, ptrdiff_t>{
        typedef list<T>::iterator it;
        list<T>* r;
    public:
        iterator(list<T>& lst,
            const typename list<T>::iterator& i)
            : it(i), r(&lst) {}
        bool operator==(const iterator& x) const {
            return it == x.it;
        }
        bool operator!=(const iterator& x) const {
            return !(*this == x);
        }
        typename list<T>::reference operator*() const {
            return *it;
        }
        iterator& operator++() {
            ++it;
            if(it == r->end())
                it = r->begin();
            return *this;
        }
        iterator operator++(int) {
            iterator tmp = *this;
            ++*this;
            return tmp;
        }
        iterator& operator--() {
            if(it == r->begin())
                it = r->end();
            --it;
            return *this;
        }
        iterator operator--(int) {
            iterator tmp = *this;
            --*this;
            return tmp;
        }
        iterator insert(const T& x) {
            return iterator(*r, r->insert(it, x));
        }
        iterator erase() {
            return iterator(*r, r->erase(it));
        }
    };
    void push_back(const T& x) { lst.push_back(x); }

```



```

    iterator begin() { return iterator(lst, lst.begin()); }
    int size() { return lst.size(); }
};

int main() {
    Ring<string> rs;
    rs.push_back("one");
    rs.push_back("two");
    rs.push_back("three");
    rs.push_back("four");
    rs.push_back("five");
    Ring<string>::iterator it = rs.begin();
    ++it; ++it;
    it.insert("six");
    it = rs.begin();
    // Twice around the ring:
    for(int i = 0; i < rs.size() * 2; i++)
        cout << *it++ << endl;
} ///:~

```

读者可以看到，绝大多数编码都是针对迭代器进行的。这个**Ring iterator**必须知道如何循环回到起始端点，所以它必须持有一个指向作为其“双亲”**Ring**对象的**list**的引用，从而知道是否已经到了环的末尾端点，这样它才能知道如何回到起始端点。

必须注意，为**Ring**设置的接口相当有限；特别是，这里没有**end()**函数，因为一个环仅仅保持进行循环的状态。这就意味着不能在需要使用超越末尾的迭代器的任何STL算法中使用**Ring**，STL中这样的算法有很多。（添加这个特征并不是无足轻重的练习。）尽管这似乎使其使用受到了限制，但是考虑一下**stack**、**queue**和**priority\_queue**，它们甚至全都没有产生任何迭代器！

## 7.14 对STL的扩充

尽管STL容器可以提供用户曾经需要的全部功能，但它们不是十全十美的。比如标准的**set**和**map**的实现都使用树型数据结构，尽管其操作相当快速，但并没有快速到足以满足用户需要的程度。在C++标准委员会中，对将利用散列算法实现的**set**和**map**包括进C++标准中的想法已经达到共识。然而由于没有足够的时间加入这些组件，最终他们放弃了这样做。<sup>①</sup>

幸运的是，还有可利用的免费替代品。有关STL的美好之处之一，就是它为创建类-STL（STL-like）的类建立了基本的模型。因此如果用户已经熟悉了STL，那么使用同样的模型创建的任何东西就都很容易理解了。

来自于Silicon Graphics<sup>®</sup>的SGI STL是最健壮的STL的实现之一，如果有需要可以用这个SGI STL替代用户编译器所使用的STL。另外，SGI增加了很多扩充的容器，包括**hash\_set**、**hash\_multiset**、**hash\_map**、**hash\_multimap**、**slist**（单链表）和**rope**（它是一个**string**的变种，对非常大型的字符串、字符串的快速联结和取子串等操作进行了优化）。

现在考虑在基于树结构的**map**和SGI **hash\_map**之间进行性能比较。为简单起见，这里将进行从**int**到**int**之间的映射：

```

//: C07:MapVsHashMap.cpp
// The hash_map header is not part of the Standard C++ STL.
// It is an extension that is only available as part of the
// SGI STL (Included with the dmc distribution).

```

① 它们可能包括在标准C++的下一个发行版本中。

② 参见 <http://www.sgi.com/tech/stl>。

```

// You can add the header by hand for all of these:
//{{-bor}{-msc}{-g++}{-mwcc}
#include <hash_map>
#include <iostream>
#include <map>
#include <ctime>
using namespace std;

int main() {
    hash_map<int, int> hm;
    map<int, int> m;
    clock_t ticks = clock();
    for(int i = 0; i < 100; i++)
        for(int j = 0; j < 1000; j++)
            m.insert(make_pair(j,j));
    cout << "map insertions: " << clock() - ticks << endl;
    ticks = clock();
    for(int i = 0; i < 100; i++)
        for(int j = 0; j < 1000; j++)
            hm.insert(make_pair(j,j));
    cout << "hash_map insertions: "
        << clock() - ticks << endl;
    ticks = clock();
    for(int i = 0; i < 100; i++)
        for(int j = 0; j < 1000; j++)
            m[j];
    cout << "map::operator[] lookups: "
        << clock() - ticks << endl;
    ticks = clock();
    for(int i = 0; i < 100; i++)
        for(int j = 0; j < 1000; j++)
            hm[j];
    cout << "hash_map::operator[] lookups: "
        << clock() - ticks << endl;
    ticks = clock();
    for(int i = 0; i < 100; i++)
        for(int j = 0; j < 1000; j++)
            m.find(j);
    cout << "map::find() lookups: "
        << clock() - ticks << endl;
    ticks = clock();
    for(int i = 0; i < 100; i++)
        for(int j = 0; j < 1000; j++)
            hm.find(j);
    cout << "hash_map::find() lookups: "
        << clock() - ticks << endl;
} ///:~

```

通过运行这个演示性能测试的程序，在所有的操作中**hash\_map**超越**map**其速度有大约4:1的改进（而且就像所预期的那样，对于两种类型的**map**进行查寻，**find()**都比**operator[]**稍微快些）。如果profiler显示出用户**map**中的性能成为系统的瓶颈，可以考虑使用**hash\_map**。

## 7.15 非STL容器

在标准库中有两种“非STL”容器：**bitset**和**valarray**。<sup>①</sup>之所以称之为“非STL”，是因为这两种容器中没有一种能够完全满足STL容器的要求。在本章前部包括了**bitset**容器，将二进制位打包成整数并且不允许对其成员进行直接寻址。**valarray**模板类是一个类**vector**的容器，

① 在前面已经提到过，在某种程度上讲**vector<bool>**特化也是一个非STL容器。

该容器对有效率的数值的计算进行了优化。这两个容器都不提供迭代器。虽然可以用非数值类型来实例化**valarray**，但是它拥有一些用于操作数值型数据的数学函数，比如**sin**、**cos**、**tan**等等。

这里有一个用来打印**valarray**中元素的工具：

```
//: C07:PrintValarray.h
#ifndef PRINTVALARRAY_H
#define PRINTVALARRAY_H
#include <valarray>
#include <iostream>
#include <cstdint>

template<class T>
void print(const char* lbl, const std::valarray<T>& a) {
    std::cout << lbl << ": ";
    for(std::size_t i = 0; i < a.size(); ++i)
        std::cout << a[i] << ' ';
    std::cout << std::endl;
}
#endif // PRINTVALARRAY_H ///:~
```

**valarray**的大多数函数和运算符都将**valarray**作为一个整体来进行操作，就像下面的例子阐明的一样：

```
//: C07:Valarray1.cpp {-bor}
// Illustrates basic valarray functionality.
#include "PrintValarray.h"
using namespace std;

double f(double x) { return 2.0 * x - 1.0; }

int main() {
    double n[] = { 1.0, 2.0, 3.0, 4.0 };
    valarray<double> v(n, sizeof n / sizeof n[0]);
    print("v", v);
    valarray<double> sh(v.shift(1));
    print("shift 1", sh);
    valarray<double> acc(v + sh);
    print("sum", acc);
    valarray<double> trig(sin(v) + cos(acc));
    print("trig", trig);
    valarray<double> p(pow(v, 3.0));
    print("3rd power", p);
    valarray<double> app(v.apply(f));
    print("f(v)", app);
    valarray<bool> eq(v == app);
    print("v == app?", eq);
    double x = v.min();
    double y = v.max();
    double z = v.sum();
    cout << "x = " << x << ", y = " << y
        << ", z = " << z << endl;
} ///:~
```

**valarray**类提供了一个构造函数，该构造函数接受一个目标类型的数组和数组中的元素计数作为其参数来初始化一个新的**valarray**。成员函数**shift( )**将每个**valarray**元素向左移动一个位置（或者，如果它的参数是个负值则向右移动），并且向移走元素后的空位中填入该类型的默认值（在这种情况下是0）。还有一个成员函数**cshift( )**，它进行循环移动（或者称为“旋转”）。所有数学运算符和函数都进行了重载以便用来操作**valarray**，二进位运算符要

求**valarray**具有相同类型和大小的参数。像**transform()**算法一样，成员函数**apply()**对每一个元素应用一个函数，但是结果被收集到一个结果**valarray**中。关系运算符返回大小匹配的**valarray<bool>**实例，该实例显示了元素与元素逐个对比的结果，例如上面的**eq**。大多数操作返回一个新的结果数组，但是很少，由于显而易见的原因，其中的一些操作返回一个数值，比如**min()**、**max()**、和**sum()**。

对**valarray**可以做的最有趣的事情就是引用其元素的一个子集，不仅可以提取信息，而且可以更新这些信息。**valarray**的一个子集被称为一个切片（slice），某些运算符使用切片来做它们的工作。下面的简单程序就使用了切片：

```
//: C07:Valarray2.cpp {-bor}{-dmc}
// Illustrates slices and masks.
#include "PrintValarray.h"
using namespace std;

int main() {
    int data[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
    valarray<int> v(data, 12);
    valarray<int> r1(v[slice(0, 4, 3)]);
    print("slice(0,4,3)", r1);
    // Extract conditionally
    valarray<int> r2(v[v > 6]);
    print("elements > 6", r2);
    // Square first column
    v[slice(0, 4, 3)] *= valarray<int>(v[slice(0, 4, 3)]);
    print("after squaring first column", v);
    // Restore it
    int idx[] = { 1, 4, 7, 10 };
    valarray<int> save(idx, 4);
    v[slice(0, 4, 3)] = save;
    print("v restored", v);
    // Extract a 2-d subset: { { 1, 3, 5 }, { 7, 9, 11 } }
    valarray<size_t> siz(2);
    siz[0] = 2;
    siz[1] = 3;
    valarray<size_t> gap(2);
    gap[0] = 6;
    gap[1] = 2;
    valarray<int> r3(v[gslice(0, siz, gap)]);
    print("2-d slice", r3);
    // Extract a subset via a boolean mask (bool elements)
    valarray<bool> mask(false, 5);
    mask[1] = mask[2] = mask[4] = true;
    valarray<int> r4(v[mask]);
    print("v[mask]", r4);
    // Extract a subset via an index mask (size_t elements)
    size_t idx2[] = { 2, 2, 3, 6 };
    valarray<size_t> mask2(idx2, 4);
    valarray<int> r5(v[mask2]);
    print("v[mask2]", r5);
    // Use an index mask in assignment
    valarray<char> text("now is the time", 15);
    valarray<char> caps("NITT", 4);
    valarray<size_t> idx3(4);
    idx3[0] = 0;
    idx3[1] = 4;
    idx3[2] = 7;
    idx3[3] = 11;
    text[idx3] = caps;
    print("capitalized", text);
} ///:~
```



一个**slice**对象接受3个参数：起始索引、要提取的元素合计数以及“跨距”，即两个用户感兴趣的元素之间的间距。切片可以用来作为一个现有**valarray**的索引，并且返回一个包含了被提取元素的新的**valarray**。比如一个**bool**型的**valarray**，它是由表达式**v>6**返回的值，也可以作为另一个**valarray**的索引；那些符合**true**值所在位置的元素都被提取出来。就像看到的那样，也可以将切片和掩码作为索引用在赋值操作的左边。一个**gslice**对象（即“generalized slice”，通用切片）就像一个切片，除了合计数和跨距参数是它们自己的数组之外，这意味着可以将一个**valarray**解释为一个多维数组。上面的例子从**v**中提取了一个2乘3的数组，从**v**中下标为0的元素开始，到相距6个元素的位置建立第1维的元素数，所做的其他事情就是在各维中每相距两个元素的位置提取一个数，这样就有效地从**v**中提取出了一个矩阵：

```
1 3 5
7 9 11
```

以下是这个程序的完整输出：

```
slice(0,4,3): 1 4 7 10
elements > 6: 7 8 9 10
after squaring v: 1 2 3 16 5 6 49 8 9 100 11 12
v restored: 1 2 3 4 5 6 7 8 9 10 11 12
2-d slice: 1 3 5 7 9 11
v[mask]: 2 3 5
v[mask2]: 3 3 4 7
capitalized: N o w   I s   T h e   T i m e
```

在矩阵乘法中可以发现一个使用切片的实际例子。考虑如何使用数组来编写两个整数矩阵相乘的函数。

```
void matmult(const int a[][MAXCOLS], size_t m, size_t n,
             const int b[][MAXCOLS], size_t p, size_t q,
             int result[][MAXCOLS]);
```

这个函数将一个**m**乘**n**的矩阵**a**和一个**p**乘**q**的矩阵**b**相乘，这里**n**和**p**应当相等。就像读者可以看到的，没有什么事情像**valarray**那样，必须为每个矩阵的第2维确定最大值，因为数组中的每个位置都是静态决定了的（固定的）。而且也很难通过值返回一个结果数组，因此调用者通常传递一个结果数组作为参数。

使用**valarray**，不仅可以传递任意大小的矩阵，而且可以容易地处理任意类型的矩阵，并且通过传值的方式返回结果。其实现方式如下所示：

```
//: C07:MatrixMultiply.cpp
// Uses valarray to multiply matrices
#include <cassert>
#include <cstdint>
#include <cmath>
#include <iostream>
#include <iomanip>
#include <valarray>
using namespace std;

// Prints a valarray as a square matrix
template<class T>
void printMatrix(const valarray<T>& a, size_t n) {
    size_t siz = n*n;
    assert(siz <= a.size());
    for(size_t i = 0; i < siz; ++i) {
        cout << setw(5) << a[i];
        cout << ((i+1)%n ? ' ' : '\n');
    }
}
```



```

    cout << endl;
}

// Multiplies compatible matrices in valarrays
template<class T>
valarray<T>
matmult(const valarray<T>& a, size_t arows, size_t acols,
        const valarray<T>& b, size_t brows, size_t bcols) {
    assert(acols == brows);
    valarray<T> result(arows * bcols);
    for(size_t i = 0; i < arows; ++i)
        for(size_t j = 0; j < bcols; ++j) {
            // Take dot product of row a[i] and col b[j]
            valarray<T> row = a[slice(acols*i, acols, 1)];
            valarray<T> col = b[slice(j, brows, bcols)];
            result[i*bcols + j] = (row * col).sum();
        }
    return result;
}

int main() {
    const int n = 3;
    int adata[n*n] = {1,0,-1,2,2,-3,3,4,0};
    int bdata[n*n] = {3,4,-1,1,-3,0,-1,1,2};
    valarray<int> a(adata, n*n);
    valarray<int> b(bdata, n*n);
    valarray<int> c(matmult(a, n, n, b, n, n));
    printMatrix(c, n);
} ///:~

```

在结果矩阵**c**中，每一个条目都是**a**中的某一行与**b**中的某一列的点积。通过使用切片，可以将这些行和列作为**valarray**提取出来，并使用全局的\*运算符和**valarray**提供的**sum()**函数进行简洁地计算。作为结果的**valarray**在运行时进行计算，没有必要担心数组维数的静态限制。在这里确实需要自行计算位置**[i][j]**的线性偏移量（参见上面的公式**i \* bcols + j**），但是为了自由地确定**valarray**的大小和类型，这是值得的。

## 7.16 小结

本章的目的不仅仅是在某种程度上深入地介绍STL容器。尽管不可能在这里涵盖STL的所有细节，读者现在也了解了足够的线索，并能在其他的资源中学习更多的信息。我们希望通过这一章帮助读者理解STL中强大的可用功能，显示了在理解和使用STL的基础上，如何能够更快速和更高效地编程。

## 7.17 练习

- 7-1 创建一个**set<char>**，打开一个文件（文件名在命令行中给出），每次从文件中读入一个**char**，将每个**char**放入该集合中。打印结果并观察其组织结构。在这个特定文件里的字母中有未被使用的字母吗？
- 7-2 创建3个**Noisy**对象序列，**vector**、**deque**和**list**。对它们进行排序。现在编写一个函数模板，接收**vector**和**deque**序列作为参数来对它们进行排序，并记录下排序的时间。编写一个特化的模板函数对**list**进行同样的操作（确保调用其成员函数**sort()**而不是使用通用算法）。比较不同类型序列的性能。
- 7-3 编写一个程序用来比较分别使用**list::sort()**以及**std::sort()**（STL算法版本的**sort()**）

对链表进行排序的速度。

- 7-4 创建一个发生器以产生0到20（包括20）之间的随机**int**型值，用它们填充一个**multiset<int>**。对每个值出现的次数进行计数，遵循例程**MultiSetWordCount.cpp**中给出的方法。
- 7-5 修改**StlShape.cpp**，让它用**deque**而不用**vector**。
- 7-6 修改**Reversible.cpp**，使其与**deque**和**list**一起工作而非**vector**。
- 7-7 使用一个**stack<int>**并将斐波那契（Fibonacci）数列存储其中。程序的命令行应该指明想要的斐波那契数列中元素的个数，还要有一个可以查看栈中是否剩下最后两个元素的循环，如果剩下最后两个元素，则在今后的每次循环中压入一个新的符合斐波那契数列的元素。
- 7-8 仅使用3个**stack**（源栈（source）、排序栈（sorted）和失败者栈（losers）），通过首先存放数字到源栈上，来对一个随机的数字序列排序。假定源栈上的栈顶元素是最大的，将其压入排序栈。持续地将源栈中的元素弹出并与排序栈中的栈顶元素比较。无论哪个栈数字最小，将最小的数字从其栈中弹出并压入失败者栈。一旦源栈为空，使用失败者栈作为源栈并重复该过程，并且使用源栈作为失败者栈。当所有的数字都已经被存入胜利者栈（排序栈）以后，算法结束。
- 7-9 打开一个文本文件，在命令行中提供其文件名。每一次从文件中读入一个单词，并使用**multiset<string>**为每个单词创建一个单词计数。
- 7-10 修改**WordCount.cpp**，使其使用**insert()**而非**operator[]**向**map**中插入元素。
- 7-11 创建拥有一个**operator<** 和一个**ostream& operator<<** 的一个类，该类应该包含一个具有优先级的数。为该类创建一个发生器，用来产生随机的具有优先级的数。用该发生器产生的数填充一个**priority\_queue**，然后取出元素并观察它们是否按照正确的顺序排列。
- 7-12 重写**Ring.cpp**，使其用一个**deque**而非**list**作为其底层实现。
- 7-13 修改**Ring.cpp**，使其底层实现可以通过模板参数来进行选择。（将那个模板参数的默认值设为**list**。）
- 7-14 创建一个名为**BitBucket**的迭代器类，它仅接收发送给它的无论什么任何东西，而不会将其写到任何地方。
- 7-15 创建一种“猜单词”的游戏程序。创建一个类，该类包含一个**char**型成员和一个指示该**char**型成员是否已经被猜中的**bool**型成员。从一个文件中随机地选择一个单词，并且将其读入用户的新类型的**vector**。重复地询问用户对一个字符的猜测，在每次猜测之后，显示该单词中已猜中的字符，对未猜中的字符显示下划线。允许给用户提供猜测全部单词的方法。在每一次猜测之后对某个值减1，在该值到达零之前如果猜中了全部单词，则用户胜出。
- 7-16 打开一个文件，并将其读入一个字符串。使该串翻转并送入一个字符串流**stringstream**。用标识符迭代器**TokenIterator**从该字符串流**stringstream**读入标识符并将其存入到**list<string>**中。
- 7-17 比较分别基于**vector**、**deque**或**list**实现的**stack**的性能。
- 7-18 创建一个模板用以实现一个名为**SList**的单链表。提供默认构造函数、**begin()** 和**end()** 函数（通过适当的嵌套迭代器），**insert()**、**erase()**和析构函数。
- 7-19 产生一个随机的整数序列，将它们存入一个**int**型数组。用其内容来初始化一个

**valarray<int>**。用**valarray**操作计算这些数字序列的和、最小值、最大值、平均值和在序列正中间元素的值。

- 7-20 用12个随机值创建一个**valarray<int>**，用20个随机值创建另一个**valarray<int>**。将第1个**valarray**理解为一个 $3 \times 4$ 的**int**型矩阵，第2个解释为 $4 \times 5$ 的**int**型矩阵，并且根据矩阵乘法的规则将它们相乘。将结果保存在大小为15的**valarray<int>**中，它表示一个 $3 \times 5$ 的结果矩阵。使用切片将第1个矩阵的行分次与第2个矩阵的列相乘。将结果以长方形的矩阵形式打印出来。



## 第三部分 专 题

专业人员的标志体现在他（或她）更加注重精益求精。在本教材的第三部分讨论C++的高级特性，以及那些被C++专业人员中的精英们所使用的开发技术。

在软件研发过程中，有时也许背离正常的面向对象设计的习语常识检查一个对象的运行时类型。大多数情况下需要用虚函数来做这项工作，但是当编写如调试器、数据库观察器或类浏览器这些特殊用途的软件工具时，则需要在运行时来决定它们的类型信息。这就是运行时类型识别（runtime type identification, RTTI）机制发挥作用的地方。RTTI是第8章的主题。

多重继承的使用在过去的这些年已经达到了滥用的地步，但某些语言甚至不支持它。当适当地运用多重继承时，它对精心制作优雅、高效的程序代码依然是一件强有力的工具。许多涉及多重继承的标准的实际应用在过去的这些年得到了长足的发展，这些内容将在第9章中介绍。

大概自从面向对象技术产生以来，在软件开发中最著名的创新就是设计模式的运用。对于在软件设计中包括的许多共同的问题，设计模式为其描述了解决方案，并且这些解决方案可以应用在许多情形中，并可以用任意一种语言来实现。在第10章中将描述许多精心挑选出的设计模式，并且用C++来实现这些设计模式。

第11章说明多线程编程的优势和所遇到的挑战。虽然大多数操作系统提供了多线程处理的功能，但标准C++现在的版本并没有说明对线程的支持。本教材使用一个可移植的、可免费利用的线程处理库来说明C++程序员可以怎样利用线程的优势去构建更多有用的和应答式的应用程序。



## 运行时类型识别

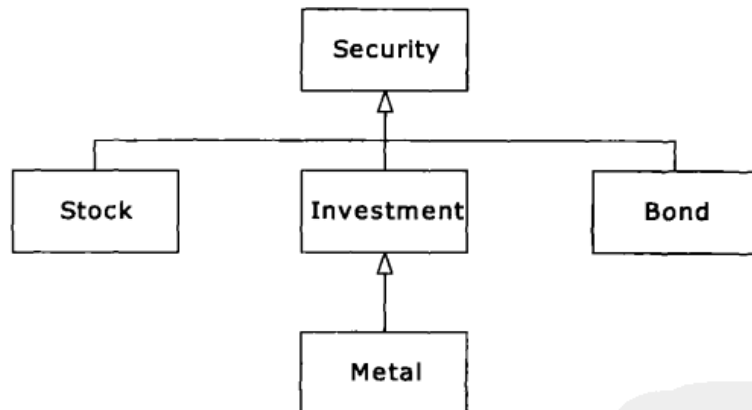
当仅有一个指针或引用指向基类型时，利用运行时类型识别（RTTI）可以找到一个对象的动态类型。

运行时类型识别可能被认为是C++中一个“次要”的特征，当程序员在编程过程中陷入非常困难的境地时，实用主义将会帮助他走出困境。正常情况下，程序员需要有意忽略对象的准确类型，而利用虚函数机制实现那个类型正确操作过程。然而，有时知道一个仅含有一个基类指针的对象的准确的运行时类型（即多半是派生的类型）是非常有用的。有了此信息，就可以更有效地进行某些特殊情况的操作，或者预防基类接口因无此信息而变得笨拙。大部分的类库都包含了虚函数，以便产生足够的运行时类型信息。当在C++中增加了异常处理时，这个特征需要对象的运行时类型的信息，因此，嵌入对这些信息的访问就使下一步工作变得很容易。本章将解释RTTI的用途和如何使用它。

### 8.1 运行时类型转换

通过指针或引用来决定对象运行时类型的一种方法是使用运行时类型转换（runtime cast），用这种方法可以查证所尝试进行的转换正确与否。当要把基类指针类型转换为派生类型时，这种方法非常有用。由于继承的层次结构的典型描述是基类在派生类之上，所以这种类型转换也称为向下类型转换（downcast）。

请看下面的类层次结构：



在下面的程序代码中，**Investment**类有一个其他类没有的额外操作，所以能够在运行时知道**Security**指针是否引用了**Investment**对象是很重要的。为了实现检查运行时的类型转换，每个类都持有一个整数标识符，以便可以与层次结构中其他的类区别开来。

```

//: C08:CheckedCast.cpp
// Checks casts at runtime.
#include <iostream>
#include <vector>
#include "../purge.h"
using namespace std;

class Security {

```

```

protected:
    enum { BASEID = 0 };
public:
    virtual ~Security() {}
    virtual bool isA(int id) { return (id == BASEID); }
};

class Stock : public Security {
    typedef Security Super;
protected:
    enum { OFFSET = 1, TYPEID = BASEID + OFFSET };
public:
    bool isA(int id) {
        return id == TYPEID || Super::isA(id);
    }
    static Stock* dynacast(Security* s) {
        return (s->isA(TYPEID)) ? static_cast<Stock*>(s) : 0;
    }
};

class Bond : public Security {
    typedef Security Super;
protected:
    enum { OFFSET = 2, TYPEID = BASEID + OFFSET };
public:
    bool isA(int id) {
        return id == TYPEID || Super::isA(id);
    }
    static Bond* dynacast(Security* s) {
        return (s->isA(TYPEID)) ? static_cast<Bond*>(s) : 0;
    }
};

class Investment : public Security {
    typedef Security Super;
protected:
    enum { OFFSET = 3, TYPEID = BASEID + OFFSET };
public:
    bool isA(int id) {
        return id == TYPEID || Super::isA(id);
    }
    static Investment* dynacast(Security* s) {
        return (s->isA(TYPEID)) ?
            static_cast<Investment*>(s) : 0;
    }
    void special() {
        cout << "special Investment function" << endl;
    }
};

class Metal : public Investment {
    typedef Investment Super;
protected:
    enum { OFFSET = 4, TYPEID = BASEID + OFFSET };
public:
    bool isA(int id) {
        return id == TYPEID || Super::isA(id);
    }
    static Metal* dynacast(Security* s) {
        return (s->isA(TYPEID)) ? static_cast<Metal*>(s) : 0;
    }
};

```



```

int main() {
    vector<Security*> portfolio;
    portfolio.push_back(new Metal);
    portfolio.push_back(new Investment);
    portfolio.push_back(new Bond);
    portfolio.push_back(new Stock);
    for(vector<Security*>::iterator it = portfolio.begin();
        it != portfolio.end(); ++it) {
        Investment* cm = Investment::dynacast(*it);
        if(cm)
            cm->special();
        else
            cout << "not an Investment" << endl;
    }
    cout << "cast from intermediate pointer:" << endl;
    Security* sp = new Metal;
    Investment* cp = Investment::dynacast(sp);
    if(cp) cout << " it's an Investment" << endl;
    Metal* mp = Metal::dynacast(sp);
    if(mp) cout << " it's a Metal too!" << endl;
    purge(portfolio);
} ///:~

```

多态的**isA()**函数检查其参数是否与它的类型参数(**id**)相容,就意味着或者**id**与对象的**typeID**准确地匹配,或者与对象的祖先之一的类型匹配(因此在这种情况下调用**Super::isA()**)。函数**dynacast()**在每个类中都是静态的,**dynacast()**为其指针参数调用**isA()**来检查类型转换是否有效。如果**isA()**返回**true**,则说明类型转换是有效的,并且返回匹配的类型转换指针。否则返回空指针,这告诉调用者类型转换无效,意味着最初的指针没有指向与想要的类型(可转换到的类型)相容的对象。对于能够检查中间类型的类型转换来说,这种机制完全是必须的,例如在前面的程序例子中,从一个指向一个**Metal**对象的**Security**类型指针,转换为**Investment**指针。<sup>①</sup>

在面向对象的应用程序中,因为平常的多态性方案解决了绝大部分问题,对大多数程序来说向下类型转换是不必要的,并且在实际的程序设计中并不提倡。然而,对于像调试器、类浏览器和数据库观察器这些工具程序来说,具有检查多派生类型转换的能力是非常重要的。借助**dynamic\_cast**操作符,C++提供这样一个可检查的类型转换。使用**dynamic\_cast**对前面的程序例子进行重写,就得到下面的程序:

```

//: C08:Security.h
#ifndef SECURITY_H
#define SECURITY_H
#include <iostream>

class Security {
public:
    virtual ~Security() {}
};

class Stock : public Security {};
class Bond : public Security {};

class Investment : public Security {
public:
    void special() {

```

① 借助微软的编译器,我们必须启用RTTI,在默认情况下这是不能使用的。启用它的命令行选项是/GR。



```

        std::cout << "special Investment function" <<std::endl;
    }
};

class Metal : public Investment {};
#endif // SECURITY_H ///:~

//: C08:CheckedCast2.cpp
// Uses RTTI's dynamic_cast.
#include <vector>
#include "../purge.h"
#include "Security.h"
using namespace std;

int main() {
    vector<Security*> portfolio;
    portfolio.push_back(new Metal);
    portfolio.push_back(new Investment);
    portfolio.push_back(new Bond);
    portfolio.push_back(new Stock);
    for(vector<Security*>::iterator it =
        portfolio.begin();
        it != portfolio.end(); ++it) {
        Investment* cm = dynamic_cast<Investment*>(*it);
        if(cm)
            cm->special();
        else
            cout << "not a Investment" << endl;
    }
    cout << "cast from intermediate pointer:" << endl;
    Security* sp = new Metal;
    Investment* cp = dynamic_cast<Investment*>(sp);
    if(cp) cout << " it's an Investment" << endl;
    Metal* mp = dynamic_cast<Metal*>(sp);
    if(mp) cout << " it's a Metal too!" << endl;
    purge(portfolio);
} ///:~

```

由于原来例子中大部分的代码开销用在了类型转换检查上，所以这个例子就变得如此之短。如同其他新式风格的C++类型转换（**static\_cast**等）一样，**dynamic\_cast**的目标类型放在一对尖括号中，并且转换对象以操作数的方式出现。如果想要安全地进行向下类型转换，**dynamic\_cast**要求使用的目标对象的类型是多态的（polymorphic）。<sup>①</sup>这就要求该类必须至少有一个虚函数。幸运的是，**Security**基类有一个虚析构函数，所以这里不需要再创建一个额外的函数去做这项工作。因为**dynamic\_cast**在程序运行时使用了虚函数表，所以比起其他新式风格的类型转换操作来说它的代价更高。

用引用而非指针同样也可以使用**dynamic\_cast**，但是由于没有诸如空引用这样的情况，这就需要采用其他方法来了解类型转换是否失败。这个“其他方法”就是捕获**bad\_cast**异常，如下所示：

```

//: C08:CatchBadCast.cpp
#include <typeinfo>
#include "Security.h"
using namespace std;

int main() {
    Metal m;

```

① 编译器典型地将一个指向一个类的RTTI表的指针插入它的虚函数表中。

```

Security& s = m;
try {
    Investment& c = dynamic_cast<Investment&>(s);
    cout << "It's an Investment" << endl;
} catch(bad_cast&) {
    cout << "s is not an Investment type" << endl;
}
try {
    Bond& b = dynamic_cast<Bond&>(s);
    cout << "It's a Bond" << endl;
} catch(bad_cast&) {
    cout << "It's not a Bond type" << endl;
}
} ///:~

```

**bad\_cast**类在<typeinfo>头文件中定义，并且像标准库的大多数的类一样，在**std**名字空间中声明。

## 8.2 typeid 操作符

获得有关一个对象运行时信息的另一个方法，就是用**typeid**操作符来完成。这种操作符返回一个**type\_info**类的对象，该对象给出与其应用有关的对象类型的信息。如果该对象的类型是多态的，它将给出那个应用（动态类型（dynamic type））的大部分派生类信息；否则，它将给出静态类型信息。**typeid**操作符的一个用途是获得一个对象的动态类型的名称，例如**const char\***，就像在下面例子中可以看到。

```

//: C08:TypeInfo.cpp
// Illustrates the typeid operator.
#include <iostream>
#include <typeinfo>
using namespace std;

struct PolyBase { virtual ~PolyBase() {} };
struct PolyDer : PolyBase { PolyDer() {} };
struct NonPolyBase {};
struct NonPolyDer : NonPolyBase { NonPolyDer(int) {} };

int main() {
    // Test polymorphic Types
    const PolyDer pd;
    const PolyBase* ppb = &pd;
    cout << typeid(ppb).name() << endl;
    cout << typeid(*ppb).name() << endl;
    cout << boolalpha << (typeid(*ppb) == typeid(pd))
        << endl;
    cout << (typeid(PolyDer) == typeid(const PolyDer))
        << endl;
    // Test non-polymorphic Types
    const NonPolyDer npd(1);
    const NonPolyBase* nppb = &npd;
    cout << typeid(nppb).name() << endl;
    cout << typeid(*nppb).name() << endl;
    cout << (typeid(*nppb) == typeid(npd)) << endl;
    // Test a built-in type
    int i;
    cout << typeid(i).name() << endl;
} ///:~

```

这个使用一个特定编译器的程序的输出是：



```

struct PolyBase const *
struct PolyDer
true
true
struct NonPolyBase const *
struct NonPolyBase
false
int

```

因为**ppb**是一个指针，所以输出的第1行是它的静态类型。为了在程序中得到RTTI的结果，需要检查指针或引用目标对象，这在第2行中说明。需要注意的是，RTTI忽略了顶层的**const**和**volatile**限定符。借助非多态类型，正好可以获得静态类型（该指针本身的类型）。正如读者所见，这里也支持内置类型。

结果是：因为没有可访问的构造函数并且禁止赋值操作，所以在**type\_info**对象中不能存储**typeid**操作的结果。必须像在演示中描述的那样来使用它。另外，通过**type\_info::name()**返回的实际字符串依赖于编译器。例如，对于一个名为**C**的类，某些编译器返回的是字符串“class C”而不是字符串“C”。把**typeid**应用到解析一个空指针的一个表达式将会引起一个**bad\_typeid**异常被抛出（该异常也定义在<typeinfo>中）。

下面的例子显示由**type\_info::name()**返回那个类名是完全限定的。

```

//: C08:RTTIandNesting.cpp
#include <iostream>
#include <typeinfo>
using namespace std;

class One {
    class Nested {};
    Nested* n;
public:
    One() : n(new Nested) {}
    ~One() { delete n; }
    Nested* nested() { return n; }
};

int main() {
    One o;
    cout << typeid(*o.nested()).name() << endl;
} ///:~

```

因为**Nested**是**One**类的一个成员类型，所以结果是**One::Nested**。

在实现定义的“整理顺序”（对文本的自然排序规则）中，也可以用**before(type\_info&)**询问一个**type\_info**对象是否在另一个**type\_info**对象之前。其返回值为**true**或**false**。当编写代码

```
if(typeid(me).before(typeid(you))) // ...
```

时，就是询问在当前的整理顺序中，**me**是否在**you**之前。如果把**type\_info**对象作为关键字会是很有用处的。

### 8.2.1 类型转换到中间层次类型

就像读者在前面使用了**Security**类层次结构的程序中所看到的，**dynamic\_cast**不仅能发现准确的类型，并且能在多层的继承层次结构中将类型转换到中间层类型。下面是另一个例子。

```

//: C08:IntermediateCast.cpp
#include <cassert>
#include <typeinfo>

```

```

using namespace std;

class B1 {
public:
    virtual ~B1() {}
};

class B2 {
public:
    virtual ~B2() {}
};

class MI : public B1, public B2 {};
class Mi2 : public MI {};

int main() {
    B2* b2 = new Mi2;
    Mi2* mi2 = dynamic_cast<Mi2*>(b2);
    MI* mi = dynamic_cast<MI*>(b2);
    B1* b1 = dynamic_cast<B1*>(b2);
    assert(typeid(b2) != typeid(Mi2*));
    assert(typeid(b2) == typeid(B2*));
    delete b2;
} ///:~

```

这个例子有关于多重继承的很复杂的情况（在本章后面部分和第9章将会学习到更多有关多重继承的知识）。如果创建一个**Mi2**对象并将它向上类型转换到该继承层次结构的根（在这种情况下，选择两个可能的根中的一个），可以成功地使**dynamic\_cast**回退至两个派生层**MI**或**Mi2**中的任何一个。

甚至可以从一个根到另一个根进行类型转换：

```
B1* b1 = dynamic_cast<B1*>(b2);
```

这也是成功的，因为**B2**实际上指向一个**Mi2**对象，该**Mi2**对象含有一个**B1**类型的子对象。

将类型转换到中间层类型，使**dynamic\_cast**和**typeid**两者之间产生一个有趣的差异。**typeid**操作符始终产生指向静态的**type\_info**型对象的引用，它描述该对象的动态类型。因此，**typeid**操作符不能给出中间层对象的类型信息。在下面的表达式中（结果是**true**），像**dynamic\_cast**一样，**typeid**并没有把**b2**当做指向派生类的指针：

```
typeid(b2) != typeid(Mi2*)
```

**b2**的类型只不过是指针类型：

```
typeid(b2) == typeid(B2*)
```

### 8.2.2 void型指针

RTTI仅仅为完整的类型工作，这就意味着当使用**typeid**时，所有的类信息都必须是可利用的。特别是，它不能与**void**型指针一起工作：

```

//: C08:VoidRTTI.cpp
// RTTI & void pointers.
//!#include <iostream>
#include <typeinfo>
using namespace std;

class Stimpy {
public:
    virtual void happy() {}
}

```

```

    virtual void joy() {}
    virtual ~Stimpy() {}
};

int main() {
    void* v = new Stimpy;
    // Error:
    //! Stimpy* s = dynamic_cast<Stimpy*>(v);
    // Error:
    //! cout << typeid(*v).name() << endl;
} ///:~

```

一个**void\***真实的意思是“无类型信息”。<sup>①</sup>

### 8.2.3 运用带模板的RTTI

因为所有的类模板所做的工作就是产生类，所以类模板可以很好地与RTTI一起工作。RTTI提供了一条方便的途径来获得对象所在类的名称。下面的示例打印出构造函数和析构函数的调用顺序：

```

//: C08:ConstructorOrder.cpp
// Order of constructor calls.
#include <iostream>
#include <typeinfo>
using namespace std;

template<int id> class Announce {
public:
    Announce() {
        cout << typeid(*this).name() << " constructor" << endl;
    }
    ~Announce() {
        cout << typeid(*this).name() << " destructor" << endl;
    }
};

class X : public Announce<0> {
    Announce<1> m1;
    Announce<2> m2;
public:
    X() { cout << "X::X()" << endl; }
    ~X() { cout << "X::~X()" << endl; }
};

int main() { X x; } ///:~

```

这个模板用一个**int**常量把一个类和其他类区分开，但是也可使用类型参数。在构造函数和析构函数内部，RTTI信息产生打印的类名。类**X**利用继承和组合两个方式创建一个类，这个类有一个有趣的构造函数和析构函数的调用顺序。输出如下：

```

Announce<0> constructor
Announce<1> constructor
Announce<2> constructor
X::X()
X::~X()
Announce<2> destructor
Announce<1> destructor
Announce<0> destructor

```

① **dynamic\_cast<void\*>**总是给出完全的对象而不是一个子对象的地址。在第9章中将更详细地解释这一点。

当然，可能会得到不同的输出结果，这取决于编译器如何表示它的**name()**信息。

### 8.3 多重继承

RTTI机制必须正确地处理多重继承的所有复杂性，包括虚基类**virtual**（在第9章将深入地进行讨论——在读过第9章之后，读者也许需要再回过头来看本节的内容）：

```
//: C08:RTTIandMultipleInheritance.cpp
#include <iostream>
#include <typeinfo>
using namespace std;

class BB {
public:
    virtual void f() {}
    virtual ~BB() {}
};

class B1 : virtual public BB {};
class B2 : virtual public BB {};
class MI : public B1, public B2 {};

int main() {
    BB* bbp = new MI; // Upcast
    // Proper name detection:
    cout << typeid(*bbp).name() << endl;
    // Dynamic_cast works properly:
    MI* mip = dynamic_cast<MI*>(bbp);
    // Can't force old-style cast:
    //! MI* mip2 = (MI*)bbp; // Compile error
} ///:-
```

**typeid()**操作符正确地检测出实际对象的名字，即便它是采用**virtual**基类指针来完成这个任务的，**dynamic\_cast**也正确地进行工作。但实际上，编译器不允许程序员用以前的方法尝试强制进行类型转换：

```
MI* mip = (MI*)bbp; // Compile-time error
```

编译器知道这样做绝不是正确的方法，因此需要程序员使用**dynamic\_cast**。

### 8.4 合理使用RTTI

因为使用RTTI能从一个匿名基类的多态指针上发现类型信息。初学者很容易误用它，因为在学会使用虚函数进行多态调用方法之前，使用RTTI很有效。对于许多有过程化编程背景的人来说，不将程序组织成**switch**语句的集合是很困难的。借助RTTI他们可以实现这个愿望，但这样就损失了多态性在代码开发和维护过程中的重要价值。C++的目的就是希望用虚函数的多态机制贯穿代码的始终，只在必须的时候使用RTTI。

然而，使用虚函数多态机制的方法调用，要求我们拥有基类定义的控制权，因为在程序扩充的某些地方，可能会发现基类并没有包含我们所需要的虚函数。如果基类来自一个库或者由别人控制，这时RTTI就是一种解决该问题的方案；可以派生一个新类，并且添加我们需要的成员函数。在程序代码的其他地方，可以检查到我们这个特定的类，并且调用它的成员函数。这样做不会破坏多态性和程序的扩展能力，因为添加这样一个新类将不需要在程序中搜索**switch**语句。然而，当需要在程序主体中增加所需的新特征的代码时，则必须使用RTTI来检查该特定的类型。

如果只是为了某个特定类的利益而在基类中放进某种新特性，这意味着由那个基类派生出的所有其他子类都为一个纯虚函数而需要保留这些毫无意义的东西。这将使接口变的更不清晰，因为我们必须覆盖由基类继承来的所有纯虚函数，这是很令人烦恼的。

最后一点，RTTI有时可以解决效率问题。如果你的程序漂亮地运用了多态性，但是某个对象是以一种极低效的方式达到这个目的的，那么就将那个类挑出来，使用RTTI，并通过为其编写特别的代码来提高效率。

### 垃圾再生器

为了更进一步地举例说明RTTI的实际用途，下面的程序模拟了一个垃圾再生器。不同种类的“垃圾”被插入一个容器中，然后根据它们的动态类型进行分类。

```

//: C08:Trash.h
// Describing trash.
#ifndef TRASH_H
#define TRASH_H
#include <iostream>

class Trash {
    float _weight;
public:
    Trash(float wt) : _weight(wt) {}
    virtual float value() const = 0;
    float weight() const { return _weight; }
    virtual ~Trash() {
        std::cout << "~Trash()" << std::endl;
    }
};

class Aluminum : public Trash {
    static float val;
public:
    Aluminum(float wt) : Trash(wt) {}
    float value() const { return val; }
    static void value(float newval) {
        val = newval;
    }
};

class Paper : public Trash {
    static float val;
public:
    Paper(float wt) : Trash(wt) {}
    float value() const { return val; }
    static void value(float newval) {
        val = newval;
    }
};

class Glass : public Trash {
    static float val;
public:
    Glass(float wt) : Trash(wt) {}
    float value() const { return val; }
    static void value(float newval) {
        val = newval;
    }
};
#endif // TRASH_H ///:~

```



用来表示垃圾类型单价的**static**值定义在实现文件中：

```
//: C08:Trash.cpp {0}
// A Trash Recycler.
#include "Trash.h"

float Aluminum::val = 1.67;
float Paper::val = 0.10;
float Glass::val = 0.23;
///:~
```

**sumValue()**模板从头到尾对一个容器进行迭代，显示并计算结果：

```
//: C08:Recycle.cpp
//{L} Trash
// A Trash Recycler.
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <typeinfo>
#include <vector>
#include "Trash.h"
#include "../purge.h"
using namespace std;

// Sums up the value of the Trash in a bin:
template<class Container>
void sumValue(Container& bin, ostream& os) {
    typename Container::iterator tally = bin.begin();
    float val = 0;
    while(tally != bin.end()) {
        val += (*tally)->weight() * (*tally)->value();
        os << "weight of " << typeid(**tally).name()
            << " = " << (*tally)->weight() << endl;
        ++tally;
    }
    os << "Total value = " << val << endl;
}

int main() {
    srand(time(0)); // Seed the random number generator
    vector<Trash*> bin;
    // Fill up the Trash bin:
    for(int i = 0; i < 30; i++)
        switch(rand() % 3) {
            case 0 :
                bin.push_back(new Aluminum((rand() % 1000)/10.0));
                break;
            case 1 :
                bin.push_back(new Paper((rand() % 1000)/10.0));
                break;
            case 2 :
                bin.push_back(new Glass((rand() % 1000)/10.0));
                break;
        }
    // Note: bins hold exact type of object, not base type:
    vector<Glass*> glassBin;
    vector<Paper*> paperBin;
    vector<Aluminum*> alumBin;
    vector<Trash*>::iterator sorter = bin.begin();
    // Sort the Trash:
    while(sorter != bin.end()) {
        Aluminum* ap = dynamic_cast<Aluminum*>(*sorter);
        Paper* pp = dynamic_cast<Paper*>(*sorter);
    }
}
```



```

    Glass* gp = dynamic_cast<Glass*>(*sorter);
    if(ap) alumBin.push_back(ap);
    else if(pp) paperBin.push_back(pp);
    else if(gp) glassBin.push_back(gp);
    ++sorter;
}
sumValue(alumBin, cout);
sumValue(paperBin, cout);
sumValue(glassBin, cout);
sumValue(bin, cout);
purge(bin);
} ///:~

```

因为垃圾被不加分类地投入一个容器中，这样一来，垃圾的所有具体类型信息就“丢失”了。但是，为了稍后适当地对废料进行分类，具体类型信息必须恢复，这将用到RTTI。

可以通过使用**map**来改进这种解决方案，该**map**将指向**type\_info**对象的指针与一个包含**Trash**指针的**vector**关联起来。因为映像需要一个能识别排序的判定函数，这里提供了一个名为**TInfoLess**的结构，它调用**type\_info::before( )**。当将**Trash**指针插入映像中的时候，这些指针将与**type\_info**关键字自动关联。注意，这里必须对**sumValue( )**进行不同的定义。

```

//: C08:Recycle2.cpp
//{L} Trash
// Recyling with a map.
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <map>
#include <typeinfo>
#include <utility>
#include <vector>
#include "Trash.h"
#include "../purge.h"
using namespace std;

// Comparator for type_info pointers
struct TInfoLess {
    bool operator()(const type_info* t1, const type_info* t2)
        const { return t1->before(*t2); }
};

typedef map<const type_info*, vector<Trash*>, TInfoLess>
    TrashMap;

// Sums up the value of the Trash in a bin:
void sumValue(const TrashMap::value_type& p, ostream& os) {
    vector<Trash*>::const_iterator tally = p.second.begin();
    float val = 0;
    while(tally != p.second.end()) {
        val += (*tally)->weight() * (*tally)->value();
        os << "weight of "
            << p.first->name() // type_info::name()
            << " = " << (*tally)->weight() << endl;
        ++tally;
    }
    os << "Total value = " << val << endl;
}

int main() {
    srand(time(0)); // Seed the random number generator

```

```

TrashMap bin;
// Fill up the Trash bin:
for(int i = 0; i < 30; i++) {
    Trash* tp;
    switch(rand() % 3) {
        case 0 :
            tp = new Aluminum((rand() % 1000)/10.0);
            break;
        case 1 :
            tp = new Paper((rand() % 1000)/10.0);
            break;
        case 2 :
            tp = new Glass((rand() % 1000)/10.0);
            break;
    }
    bin[&typeid(*tp)].push_back(tp);
}
// Print sorted results
for(TrashMap::iterator p = bin.begin();
    p != bin.end(); ++p) {
    sumValue(*p, cout);
    purge(p->second);
}
} ///:~

```

为了直接调用**type\_info::name()**，我们在这里修改了**sumValue()**，因为作为**TrashMap::value\_type**对的第1个成员，**type\_info**对象现在是可获得的。这样就避免了为了获得正在处理的**Trash**的类型名而额外调用**typeid**，而这在该程序的以前版本中却是必须做的。

## 8.5 RTTI的机制和开销

实现RTTI典型的方法是，通过在类的虚函数表中放置一个附加的指针。这个指针指向那个特别类型的**type\_info**结构。**typeid()**表达式的结果非常简单：虚函数表指针取得**type\_info**指针，并且产生一个对**type\_info**结构的引用。因为这正好是一个双指针的解析操作，这是一个代价为常量时间的操作。

对于**dynamic\_cast<destination\*>(source\_pointer)**来说，大部分的情况是相当直观的：检索**source\_pointer**的RTTI信息，并且为**destination\***类型取得RTTI信息。然后，库程序确定**source\_pointer**类型是否属于类型**destination\***或**destination\***的一个基类。如果该基类型不是派生类的第1个基类，那么由于多重继承的原因返回的指针将是被调整过的。在继承层次结构和虚基类的使用中，因为一个基类型可以出现多次，所以对于多重继承来说情况将会更加复杂。

因为为了**dynamic\_cast**而使用的库程序必须从头至尾对一个基类表进行检查，**dynamic\_cast**开销可能高于**typeid()**（但是分别获得了不同的信息，这些信息对于问题的解决来说是必要的），找到一个基类比找到一个派生类可能需要花更多的时间。另外，**dynamic\_cast**将任何一个类型与任何其他类型相比较；在同一层次结构中可以不受限制地进行类型比较。这就增加了由**dynamic\_cast**使用的库程序的额外开销。

## 8.6 小结

尽管通常情况下会为一个指向其基类的指针进行向上类型转换，然后再使用那个基类的通用接口（通过虚函数），但是如果知道一个由基类指针指向的对象的动态类型，有时候根据获

得的这些信息进行相关处理可能会使事情变得更加有效，而这些正是RTTI所提供的。大部分通常的误用来自一些程序员，这些误用是由于他们不理解虚函数而是采用RTTI来做类型检查的编码所造成的。C++的基本原理似乎提供了对违反类型的定义规则和完整性的情况进行监督和纠正的强有力的工具和保护，但是如果有谁想故意地误用或回避某一个语言的特征，那么将没有什么人可以阻止他这样做。有时候误用导致的小错却是取得经验的最快方法。

## 8.7 练习

- 8-1 创建一个类**Base**，它有一个**virtual**虚析构函数，同时创建一个派生类**Derived**，它派生于类**Base**。创建一个存储**Base**指针的**vector**，该指针指向随机生成的**Base**和**Derived**对象。使用该**vector**的内容来填充另外一个包含所有**Derived**指针的另一个**vector**。比较**typeid()**和**dynamic\_cast**的执行时间，看哪一个执行得更快。
- 8-2 修改本教材第1卷中的**C16:AutoCounter.h**，使它成为一个有用的调试工具。它将作为那些与追踪有关的各个类的嵌套成员来使用。将**AutoCounter**修改为一个模板，它将外围类的类名作为模板的参数，并且在所有的出错信息中利用RTTI来打印类名。
- 8-3 通过使用**typeid()**打印出模板的准确的名称，用RTTI作为辅助工具进行程序的调试。实例化各种类型的模板，并看看它们的结果是什么。
- 8-4 通过先将**Wind5.cpp**复制到一个新位置，修改第1卷第14章中的**Instrument**的层次结构。现在，在**Wind**类中新增加一个虚函数**clearSpitValve()**，并且在继承自**Wind**的所有派生类中重新定义它。实例化一个存储**Instrument**指针的**vector**，并用**new**操作符创建各种类型的**Instrument**对象来填充它。现在使用RTTI在这样一个容器中遍历查找类**Wind**或**Wind**的派生类的对象。并对这些对象调用**clearSpitValve()**函数。注意，如果在工具（**Instrument**）基类中含有一个**clearSpitValve()**函数，那么将会使该基类发生使人不愉快的混乱。
- 8-5 修改上一个练习，在该基类中放置一个**prepareInstrument()**函数，它需要调用适当的函数（例如，在它适宜的时候调用**clearSpitValve()**）。注意，**prepareInstrument()**是放置在基类中的一个明智的函数，它的使用剔除了在上一个练习题中对RTTI的需要。
- 8-6 创建一个含有指针的**vector**，这些指针指向10个随机**Shape**对象（例如，至少是若干个**Square**和**Circle**）。重写每个具体的类中的**draw()**成员函数，用于打印输出被画对象的尺寸（任何一个应用的长度或半径）。编写一个**main()**程序，首先画出容器中所有的**Square**，并按其长度进行排序，然后再画出所有的**Circle**，并按其半径进行排序。
- 8-7 创建一个大的**vector**，它存储那些指向随机**Shape**对象的指针。在**Shape**中编写一个非虚（non-virtual）**draw()**函数，使用RTTI来确定每个对象的动态类型，并且借助开关（switch）语句执行适当的代码来“画出”对象。然后使用虚函数，“用正确的方法”重新编写**Shape**的层次结构。比较两种方法的实现代码长度和执行时间。
- 8-8 创建一个关于**Pet**类的层次结构，其中包括**Dog**、**Cat**和**Horse**。再创建一个关于**Food**类的层次结构：其中包括**Beef**、**Fish**和**Oats**。**Dog**类有一个成员函数**eat()**，其参数为**Beef**，同样**Cat::eat()**将**Fish**对象作为其参数，而**Oats**对象则作为参数传递给**Horse::eat()**。创建这样一个**vector**，它含有指向随机生成的**Pet**对象的指针，并且访问每一个**Pet**，并将正确的**Food**对象类型传递给对应的**eat()**函数。

- 8-9 建立一个名为**drawQuad()**的全局函数，它使用一个**Shape**对象的引用作为参数。如果它有4条边（也就是说，它是**Square**或**Rectangle**），那么它将调用其含有**Shape**参数的**draw()**函数。否则将打印消息“不是一个四边形”。遍历这个包含指向随机生成的**Shape**对象指针的**vector**，在遍历时对每个被访问对象调用**drawQuad()**。在**vector**中，放置那些指向**Square**、**Rectangle**、**Circle**和**Triangle**对象的指针。
- 8-10 根据类名对一个含有随机**Shape**对象的**vector**排序。用**type\_info::before()**作为排序的比较函数。



## 多重继承

多重继承 (MI) 的基本概念听起来相当简单：通过继承多个基类来创建一个新类。确切地说这种多重继承语法正是我们所期望的，并且只要继承层次结构图是简单的，那么多重继承也同样简单。

尽管MI可能引入一些二义性和奇怪的案例，在本章将对这些案例进行讨论。但是首先，这些案例将有助于读者对该主题获得一些基本认识。

### 9.1 概论

在C++之前，最成功的面向对象的语言是Smalltalk。Smalltalk是作为一种完全的面向对象语言而创造出来的。它被称作是纯粹的 (pure) 面向对象语言，而C++则被称作是一种混合的 (hybrid) 语言，这是因为C++支持多种形式的程序设计范例，而不仅仅只是面向对象的程序设计范例。一个由Smalltalk做出的设计本身就决定了所有类都是在一个单一的继承层次结构中派生的，都以一个基类作为根 (称为**Object**——这就是基于对象的继承层次结构 (object-based hierarchy) 模型)。<sup>①</sup>在Smalltalk中，不可能创建这样一个新类：它不是派生自一个现存的类。这就是为什么在Smalltalk中实现多种形式的继承方式要花费大量的时间：在开始建立新类前，必须学习和掌握类库。因此Smalltalk的类继承层次结构是一棵单一的整体树。

Smalltalk中的类通常有很多的共同点，并且总是有某些共通的东西 (**Object**的特征和行为)，所以不会经常遇上需要从多个基类继承的情况。然而，在C++中却可以建立用户想要的多种不同的继承树。所以为了逻辑上的完整性，该语言必须有能力一次组合多个类——因而需要多重继承。

然而，程序员对多重继承的需求并不是显而易见的。关于在C++中多重继承是否是必要的问题存在着 (现在仍然存在着) 大量的争论。1989年在AT&T **cf**ront 发布版 (release) 2.0中加入了MI，这也是C++语言1.0版以来发生的首次重要的变化。<sup>②</sup>从那以后，许多其他的特征被加入标准C++中 (最著名的是模板)，这些变化改变了编程的思想并且使MI的作用处于次要的地位。程序设计人员可以把MI看做是一个“次要”的语言特征，也就是说，在日常的程序设计决定中很少涉及它。

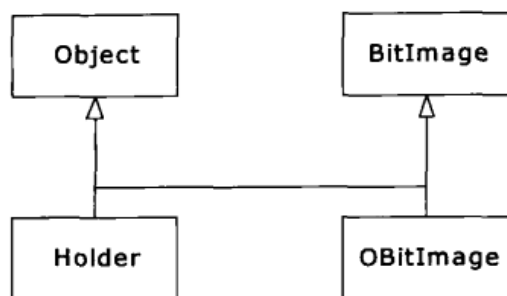
有关MI最激烈的争论之一涉及容器。假如要想建立这样一个容器，每个人都可以很容易地使用它。一种方法是将**void\***作为该容器内部的类型。然而Smalltalk的方法是建立一个持有**Object**对象的容器，因为**Object**是Smalltalk继承层次结构的基类型。Smalltalk中的所有内容最终都派生自**Object**，所以持有**Object**的容器可以存储任何类型的对象。

现在考虑在C++中的情况。假设供应商A建立了一个基于对象的继承层次结构，该继承层次结构包括了一组有用的容器，这些容器中就包含想要使用的一种称为**Holder**的容器。接下来偶然遇到了供应商B提供的类继承层次结构，它包含了其他一些比较重要的类，例如**BitImage**类，它持有生动的图像。制造一个持有这些**BitImage**特征和行为的**Holder**容器的惟一方法，是创建一个派生自**Object**和**BitImage**两者的新类，这样，在**Holder**中就可以

① 对Java和其他面向对象的语言来说这也是正确的。

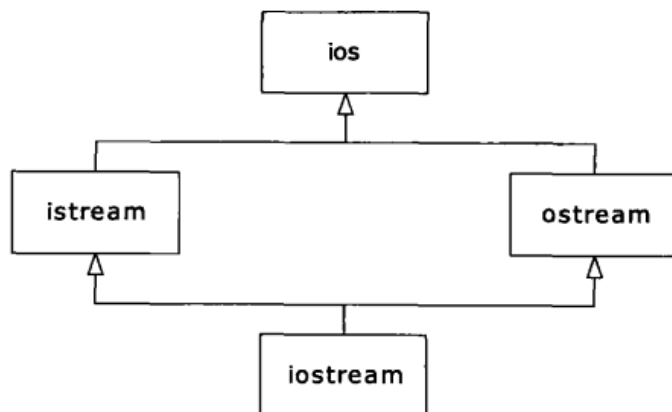
② 这些版本号是国际AT&T的编号方式。

持有**BitImage**中的特征和行为:



这似乎是需要MI的一个重要理由，而许多类库就是建立在这种模型之上的。然而如第5章所述，模板的加入改变了创建容器的方法。所以这种情况不再是使用MI的动力。

而需要MI的另外一个原因跟设计有关。可以有意地用MI来使一个程序设计得更加灵活或更实用（至少表面上是这样）。在原始的**iostream**库设计中就有这样的一个例子（仍存在于目前的模板设计中，如第4章所述）：



**istream**和**ostream**就其自身来说都是有用的类，但是也可以通过从一个类同时派生出这两个类的方式产生它们，而该基类将这两个类的特征和行为结合在一起。类**ios**提供了所有这些流类的共同点，在这种情况下MI就是一种代码分解机制。

不管是什么原因激发我们使用MI，但是要真正使用它将比看上去要难得多。

## 9.2 接口继承

多重继承中毫无争议的一种运用属于接口继承（interface inheritance）。在C++中，所有的继承都是实现继承（implementation inheritance），因为在一个基类、接口和实现中的任何内容都将成为派生类的一部分。只继承一个类的某些部分（比如只继承接口）是不可能的。就像第1卷第14章说明的那样，当客户使用派生类的对象时，私有的和被保护的继承将可能限制派生类成员对基类成员的访问，但是这些并不会影响派生类；它仍然包含了所有的基类数据，并且可以访问所有的非私有的基类成员。

另一方面，接口继承仅仅是在一个派生类接口中加入了成员函数的声明（declaration），在C++中并不直接支持这种使用方法。C++中模拟接口继承常见的技术是从一个仅包含声明（没有数据和函数体）的接口类（interface class）派生一个类。除了析构函数以外，这些声明都是纯虚函数。举例如下：

```

//: C09:Interfaces.cpp
// Multiple interface inheritance.
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

class Printable {
public:
    virtual ~Printable() {}
    virtual void print(ostream&) const = 0;
};

class Intable {
public:
    virtual ~Intable() {}
    virtual int toInt() const = 0;
};

class Stringable {
public:
    virtual ~Stringable() {}
    virtual string toString() const = 0;
};

class Able : public Printable, public Intable,
             public Stringable {
    int myData;
public:
    Able(int x) { myData = x; }
    void print(ostream& os) const { os << myData; }
    int toInt() const { return myData; }
    string toString() const {
        ostringstream os;
        os << myData;
        return os.str();
    }
};

void testPrintable(const Printable& p) {
    p.print(cout);
    cout << endl;
}

void testIntable(const Intable& n) {
    cout << n.toInt() + 1 << endl;
}

void testStringable(const Stringable& s) {
    cout << s.toString() + "th" << endl;
}

int main() {
    Able a(7);
    testPrintable(a);
    testIntable(a);
    testStringable(a);
} ///:~

```

类**Able**“实现”了接口**Printable**、**Intable**和**Stringable**，因为它提供了那些对它们进行声明的函数的实现。因为**Able**派生自所有这3个类，**Able**对象具有多“is-a”关系。例如，对象**a**的行为可能像一个**Printable**对象，因为它的类**Able**公有派生自**Printable**，并且提供

了对`print()`的实现。测试函数并不需要知道作为参数使用的大多数的派生类对象的类型；它仅仅需要这样一个可以代替它们的参数类型的对象。

一般来说，采用模板来解决问题的方法将会使程序变得更加简洁：

```
//: C09:Interfaces2.cpp
// Implicit interface inheritance via templates.
#include <iostream>
#include <sstream>
#include <string>
using namespace std;
class Able {
    int myData;
public:
    Able(int x) { myData = x; }
    void print(ostream& os) const { os << myData; }
    int toInt() const { return myData; }
    string toString() const {
        ostringstream os;
        os << myData;
        return os.str();
    }
};

template<class Printable>
void testPrintable(const Printable& p) {
    p.print(cout);
    cout << endl;
}

template<class Intable>
void testIntable(const Intable& n) {
    cout << n.toInt() + 1 << endl;
}

template<class Stringable>
void testStringable(const Stringable& s) {
    cout << s.toString() + "th" << endl;
}

int main() {
    Able a(7);
    testPrintable(a);
    testIntable(a);
    testStringable(a);
} ///:~
```

**Printable**、**Intable**和**Stringable**这些名字现在仅是模板参数，这些参数假设在各自的语境中表示存在的操作。换句话说，测试函数可以接受任何一种类型的参数，这些参数类型与正确的识别标志和返回类型一起提供了一个成员函数的定义；这些参数并不必要派生自一个共同的基类。有些人更适应第1种版本的示例，因为类型名可以通过继承关系保证能够确定，该继承关系由预期的接口来实现。而其他人更满足于这样一个事实，如果提供的模板类型参数不能满足测试函数所需要的操作，该错误在编译时仍然会被捕获。对比前一个方法（继承），后面的方法是一种“较弱”的类型检查形式，但是对程序员（和程序）来说，效果是相同的。这就是被许多现代C++程序员所接受的弱输入检查的一种形式。

### 9.3 实现继承

如前所述，C++仅仅提供了实现继承，这就意味着所有的内容总是继承自基类。这样做有



很大的好处，因为它将使程序员从不得不在派生类中实现所有的细节（正如前面的例子中所采用接口继承所做的事情）中解放出来。多重继承的一个共同用途包括使用混入类（mixin），这些混入类的存在是为了通过继承来增加其他类的功能。混入类不能刻意地由它本身进行实例化。

举个例子，假设一个客户使用了某个类，该类支持访问一个数据库。在这个情况下，仅仅有一个头文件可以使用——在这里指出，客户不能访问实现具体功能的这部分源代码。举例说明，假定**Database**类的实现如下所示：

```
//: C09:Database.h
// A prototypical resource class.
#ifndef DATABASE_H
#define DATABASE_H
#include <iostream>
#include <stdexcept>
#include <string>

struct DatabaseError : std::runtime_error {
    DatabaseError(const std::string& msg)
        : std::runtime_error(msg) {}
};

class Database {
    std::string dbid;
public:
    Database(const std::string& dbStr) : dbid(dbStr) {}
    virtual ~Database() {}
    void open() throw(DatabaseError) {
        std::cout << "Connected to " << dbid << std::endl;
    }
    void close() {
        std::cout << dbid << " closed" << std::endl;
    }
    // Other database functions...
};
#endif // DATABASE_H ///:~
```

这里已经省略了实际的数据库功能（存储操作、检索操作，等等），但是在这里那些功能并不重要。使用这个类需要一个数据库连接串，并调用**Database::open()**来连接数据库，通过调用**Database::close()**断开连接：

```
//: C09:UseDatabase.cpp
#include "Database.h"

int main() {
    Database db("MyDatabase");
    db.open();
    // Use other db functions...
    db.close();
}
/* Output:
connected to MyDatabase
MyDatabase closed
*/ ///:~
```

在一个典型的客户机-服务器模式的情况下，客户拥有多个对象，这些对象分享一个连接的数据库。尽管数据库的最后关闭是非常重要的，但数据库只能在不再需要访问它之后关闭。通常，将这种行为封装到一个类中，用来实现对使用数据库连接的客户实体的数目进行跟踪，并且在实体计数归为零时自动终止数据库的连接。为了给**Database**类加入引用计数，利用多重继

承将一个叫**Countable**的类混入**Database**类中，这样就创建了一个新类**DBConnection**。这就是**Countable**混入类：

```
//: C09:Countable.h
// A "mixin" class.
#ifndef COUNTABLE_H
#define COUNTABLE_H
#include <cassert>

class Countable {
    long count;
protected:
    Countable() { count = 0; }
    virtual ~Countable() { assert(count == 0); }
public:
    long attach() { return ++count; }
    long detach() {
        return (--count > 0) ? count : (delete this, 0);
    }
    long refCount() const { return count; }
};
#endif // COUNTABLE_H ///:~
```

很明显，这不是一个独立类，因为它的构造函数是**protected**类型；它需要一个友元或派生类来使用它。析构函数是虚函数这一点非常重要，因为它只被**detach()**中的**delete this**语句调用，并且需要将派生对象正确地销毁。<sup>⊖</sup>

**DBConnection**类继承了**Database**和**Countable**，并且提供了一个静态的**create()**函数，这个函数用来初始化它的**Countable**子对象。这是将在第10章中讨论的工厂方法(Factory Method)设计模式的一个例子：

```
//: C09:DBConnection.h
// Uses a "mixin" class.
#ifndef DBCONNECTION_H
#define DBCONNECTION_H
#include <cassert>
#include <string>
#include "Countable.h"
#include "Database.h"
using std::string;

class DBConnection : public Database, public Countable {
    DBConnection(const DBConnection&); // Disallow copy
    DBConnection& operator=(const DBConnection&);
protected:
    DBConnection(const string& dbStr) throw(DatabaseError)
        : Database(dbStr) { open(); }
    ~DBConnection() { close(); }
public:
    static DBConnection*
    create(const string& dbStr) throw(DatabaseError) {
        DBConnection* con = new DBConnection(dbStr);
        con->attach();
        assert(con->refCount() == 1);
        return con;
    }
    // Other added functionality as desired...
};
```

⊖ 尽管这很重要，但是我们不需要未定义的行为。对一个基类来说没有一个虚析构函数将是一个错误。

```
};
#endif // DBCONNECTION_H ///:~
```

不用修改**Database**类，现在就有一个引用计数的数据库连接，并且可以确保数据库连接不会被偷偷地终止。通过**DBConnection**的构造函数和析构函数，使用第1章中提到的资源获取式初始化（the Resource Acquisition Is Initialization, RAII）方法来实现数据库的打开和关闭。这就使得**DBConnection**的使用变得很容易：

```
//: C09:UseDatabase2.cpp
// Tests the Countable "mixin" class.
#include <cassert>
#include "DBConnection.h"

class DBClient {
    DBConnection* db;
public:
    DBClient(DBConnection* dbCon) {
        db = dbCon;
        db->attach();
    }
    ~DBClient() { db->detach(); }
    // Other database requests using db...
};

int main() {
    DBConnection* db = DBConnection::create("MyDatabase");
    assert(db->refCount() == 1);
    DBClient c1(db);
    assert(db->refCount() == 2);
    DBClient c2(db);
    assert(db->refCount() == 3);
    // Use database, then release attach from original create
    db->detach();
    assert(db->refCount() == 2);
} ///:~
```

因为对**DBConnection::create( )**的调用又调用了**attach( )**，所以在结束时，必须显式调用**detach( )**来释放数据库的初始连接。注意，**DBClient**类也用RAII管理连接的使用。当程序结束时，这两个**DBClient**对象的析构函数将分别使引用计数减1（通过调用**detach( )**完成，这里的**DBConnection**继承自**Countable**），在对象**c1**被销毁后，当引用计数达到零时数据库连接将被关闭（因为调用了**Countable**的虚析构函数）。

模板方法一般用于混入继承，允许用户在编译时指定想要的混入类的类型。这样就可以使用不同的引用计数方法来完成这项工作，而不用显式地两次定义**DBConnection**。下面这个例子说明了这种方法是如何工作的：

```
//: C09:DBConnection2.h
// A parameterized mixin.
#ifndef DBCONNECTION2_H
#define DBCONNECTION2_H
#include <cassert>
#include <string>
#include "Database.h"
using std::string;

template<class Counter>
class DBConnection : public Database, public Counter {
    DBConnection(const DBConnection&); // Disallow copy
    DBConnection& operator=(const DBConnection&);
```

```
protected:
    DBConnection(const string& dbStr) throw(DatabaseError)
    : Database(dbStr) { open(); }
    ~DBConnection() { close(); }
public:
    static DBConnection* create(const string& dbStr)
    throw(DatabaseError) {
        DBConnection* con = new DBConnection(dbStr);
        con->attach();
        assert(con->refCount() == 1);
        return con;
    }
    // Other added functionality as desired...
};
#endif // DBCONNECTION2_H ///:~
```

这里惟一的变化是用于类定义的模板前缀（以及为了清楚起见而将**Countable**重新命名为**Counter**）。也可以把某个数据库类作为一个模板参数（可以从多个数据库访问类中进行选择），但它并不是一个混入类，因为它是一个独立类。尽管下面的例子将原始的**Countable**作为**Counter**混入类型使用，但是可以使用实现了适当的接口（**attach()**、**detach()**等等）的任何类型。

```
///C09:UseDatabase3.cpp
// Tests a parameterized "mixin" class.
#include <cassert>
#include "Countable.h"
#include "DBConnection2.h"

class DBClient {
    DBConnection<Countable>* db;
public:
    DBClient(DBConnection<Countable>* dbCon) {
        db = dbCon;
        db->attach();
    }
    ~DBClient() { db->detach(); }
};

int main() {
    DBConnection<Countable>* db =
        DBConnection<Countable>::create("MyDatabase");
    assert(db->refCount() == 1);
    DBClient c1(db);
    assert(db->refCount() == 2);
    DBClient c2(db);
    assert(db->refCount() == 3);
    db->detach();
    assert(db->refCount() == 2);
} ///:~
```

多参数混入类型的一般模式很简单：

```
template<class Mixin1, class Mixin2, ..., class MixinK>
class Subject : public Mixin1,
               public Mixin2,
               ...,
               public MixinK {...};
```

## 9.4 重复子对象

当从某个基类继承时，可以在其派生类中得到那个基类的所有数据成员的副本。下面的程

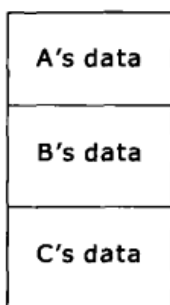
序说明了多个基类子对象在内存中的可能布局：<sup>①</sup>

```
//: C09:Offset.cpp
// Illustrates layout of subobjects with MI.
#include <iostream>
using namespace std;

class A { int x; };
class B { int y; };
class C : public A, public B { int z; };

int main() {
    cout << "sizeof(A) == " << sizeof(A) << endl;
    cout << "sizeof(B) == " << sizeof(B) << endl;
    cout << "sizeof(C) == " << sizeof(C) << endl;
    C c;
    cout << "&c == " << &c << endl;
    A* ap = &c;
    B* bp = &c;
    cout << "ap == " << static_cast<void*>(ap) << endl;
    cout << "bp == " << static_cast<void*>(bp) << endl;
    C* cp = static_cast<C*>(bp);
    cout << "cp == " << static_cast<void*>(cp) << endl;
    cout << "bp == cp? " << boolalpha << (bp == cp) << endl;
    cp = 0;
    bp = cp;
    cout << bp << endl;
}
/* Output:
sizeof(A) == 4
sizeof(B) == 4
sizeof(C) == 12
&c == 1245052
ap == 1245052
bp == 1245056
cp == 1245052
bp == cp? true
0
*/ ///:~
```

正如读者所见，对象**c**的**B**子对象部分从整个对象开始位置偏移了4个字节。其布局如下所示：



对象**c**以它的**A**子对象作为开头，然后是**B**子对象部分，最后的数据完全来自类型**C**本身。因为**C**“is-an”<sup>②</sup>**A**并且也“is-a”**B**，所以它可以向上类型转换为两者之中任一基类型。当向

<sup>①</sup> 实际的布局在实现时确定。

<sup>②</sup> “我们常把基类和派生类之间的关系看做是一个‘is-a’（是）关系”。见《C++编程思想第1卷：标准C++导引》第1章。——编辑注

上类型转换为**A**时，结果指针指向**A**子对象部分，这发生在**C**对象的开始位置，所以地址**ap**等同于表达式**&c**。然而，当向上类型转换为**B**子对象时，结果指针必须指向**B**子对象所在的实际位置，因为类**B**并不知道有关类**C**（或类**A**，就本例而言）的事情。换句话说，被**bp**指向的对象必须能够产生和独立的**B**对象相同的行为（除了任何需要多态的行为以外）。

当对**bp**进行类型转换倒退为一个**C\***时，由于原始对象是**C**，**bp**指向该对象开始的位置，因为它已经知道**B**子对象的位置，所以指针被调整指向了完整对象的起始地址。如果**bp**指向的是一个独立的**B**对象而不是**C**对象的开始位置，那么这种类型转换就是不合法的。<sup>①</sup>此外，在比较表达式**bp == cp**中，**cp**被隐式转换为**B\***，这是使该比较表达式变得有意义的惟一方法（这就是说，向上类型转换总是允许的），因此结果是**true**。因此，当在子对象和完整类型间来回转换时，要应用适当的偏移量。

空指针需要进行特别的处理。显然，如果在开始进行类型转换时指针为零，那么在转换到一个**B**子对象或从一个**B**子对象转换回来时，由于盲目地减去偏移量将会导致产生无效的地址。基于这种原因，当类型转换到**B\***或有来自**B\***的类型转换时，编译器产生逻辑检查，首先查看该指针是否为零。如果不为零，则可以应用偏移量；否则，当指针为零时就放弃使用偏移量。

根据目前学习到的语法，如果现在有多个基类，并且如果这些基类依次有一个共同的基类，那么将得到顶层基类的两个副本。如下面的例子所示：

```
//: C09:Duplicate.cpp
// Shows duplicate subobjects.
#include <iostream>
using namespace std;

class Top {
    int x;
public:
    Top(int n) { x = n; }
};

class Left : public Top {
    int y;
public:
    Left(int m, int n) : Top(m) { y = n; }
};

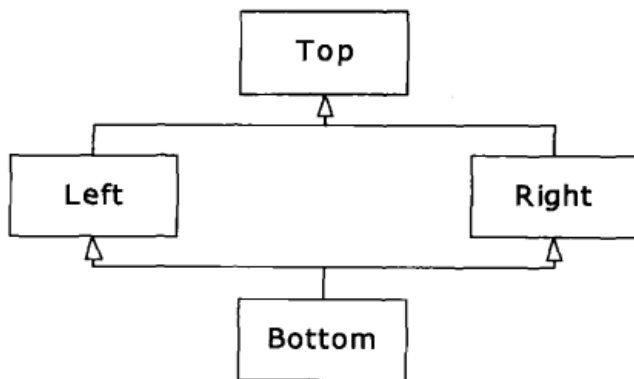
class Right : public Top {
    int z;
public:
    Right(int m, int n) : Top(m) { z = n; }
};

class Bottom : public Left, public Right {
    int w;
public:
    Bottom(int i, int j, int k, int m)
        : Left(i, k), Right(j, k) { w = m; }
};

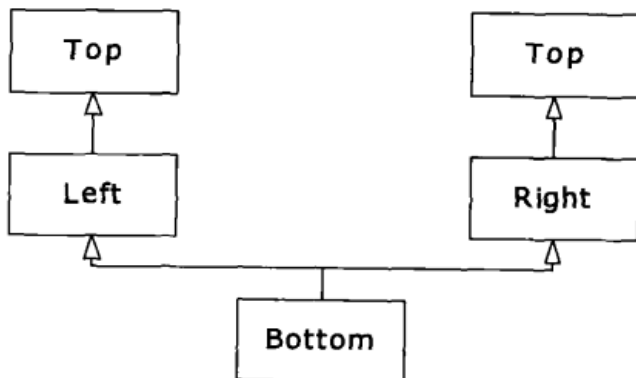
int main() {
    Bottom b(1, 2, 3, 4);
    cout << sizeof b << endl; // 20
} ///:~
```

① 但并不作为一个错误被检查。**dynamic\_cast**可以解决这个问题。看前面章节的详细说明。

因为对象**b**的长度是20个字节，<sup>①</sup>所以在一个完整的**Bottom**对象中共有5个整型变量。这种情况的典型类图通常如下所示：



这就是所谓的“菱形继承”（也称“钻石继承”），但是在这个例子中可以较好地表示为如下的类图：



这种设计的不足之处表现在前面代码中**Bottom**类的构造函数上。用户认为只需要4个整型变量，但是哪些实际参数才是传递给**Left**和**Right**所需要的两个参数呢？尽管这一设计并不是固有的“错误”，但通常它并不是一个应用程序所需要的。在尝试将指向**Bottom**对象的指针转换成指向**Top**的指针时，同样也会出现问题。如前所述，可能需要调整对象指针的地址，这依赖于在完整的对象内部各子对象所处的位置，但是这里却有两个**Top**子对象供选择。因为编译器不知道选择哪一个，所以这样一种向上类型转换是模棱两可的（二义性），也是不允许的。用同样的原因可以解释为什么一个**Bottom**对象不能调用那个只定义在**Top**中的函数。如果存在这样一个函数**Top::f()**，那么调用**b.f()**需要涉及一个**Top**子对象作为执行语境，而这里却有两个**Top**可供选择。

## 9.5 虚基类

在这种情况下通常需要的是真正的菱形继承，**Left**和**Right**子对象在一个完整的**Bottom**对象内部共享着一个**Top**对象，这正是第1个类图描述的情况。这是通过使**Top**成为**Left**和**Right**的一个虚基类（virtual base class）来完成的：

① 即 $5 * \text{sizeof}(\text{int})$ 。因为编译器可以加入任意的数据类型，所以对象的长度至少是它各部分的总和，也可以更长。

```

//: C09:VirtualBase.cpp
// Shows a shared subobject via a virtual base.
#include <iostream>
using namespace std;
class Top {
protected:
    int x;
public:
    Top(int n) { x = n; }
    virtual ~Top() {}
    friend ostream&
    operator<<(ostream& os, const Top& t) {
        return os << t.x;
    }
};

class Left : virtual public Top {
protected:
    int y;
public:
    Left(int m, int n) : Top(m) { y = n; }
};

class Right : virtual public Top {
protected:
    int z;
public:
    Right(int m, int n) : Top(m) { z = n; }
};

class Bottom : public Left, public Right {
    int w;
public:
    Bottom(int i, int j, int k, int m)
        : Top(i), Left(0, j), Right(0, k) { w = m; }
    friend ostream&
    operator<<(ostream& os, const Bottom& b) {
        return os << b.x << ', ' << b.y << ', ' << b.z
            << ', ' << b.w;
    }
};

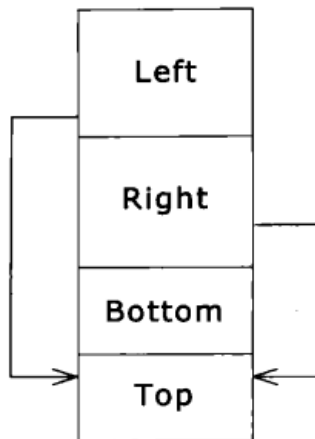
int main() {
    Bottom b(1, 2, 3, 4);
    cout << sizeof b << endl;
    cout << b << endl;
    cout << static_cast<void*>(&b) << endl;
    Top* p = static_cast<Top*>(&b);
    cout << *p << endl;
    cout << static_cast<void*>(p) << endl;
    cout << dynamic_cast<void*>(p) << endl;
} ///:~

```

给定类型的各个虚基类都涉及相同的对象，不论它在层次结构的哪个地方出现。<sup>①</sup>这意味着，当一个**Bottom**对象被实例化时，对象的布局看起来像下面的样子：

① 使用术语层次结构 (hierarchy) 因为人人都在使用它，但是用来表示多重继承关系的图一般是一个有向无环图 (DAG)，也称为网格，其理由是显而易见的。





**Left**和**Right**子对象各有一个指向共享的**Top**子对象的指针（或某种概念上等价的对象），并且对**Left**和**Right**成员函数中那个子对象的所有引用都要通过这些指针来完成。<sup>①</sup>在这里，当从一个**Bottom**向上类型转换为一个**Top**对象时，不存在二义性的问题，因为这里只有一个**Top**对象可用来进行转换。

前面程序的输出结果如下：

```

36
1,2,3,4
1245032
1
1245060
1245032

```

打印出来的地址说明这种特殊的实现确实在完整的对象的结尾处储存**Top**子对象（尽管实际上它在那里并不重要）。**dynamic\_cast**到**void\***的结果总是确定指向完整对象的地址。

尽管在技术上这样做是不合法的，<sup>②</sup>但是如果去掉了虚析构函数（和**dynamic\_cast**语句，这样程序将可以通过编译），那么**Bottom**的长度将减少到24个字节。似乎**Bottom**的长度的减少量正好等于3个指针的大小。为什么呢？

重要的是不必太按照字面上的意思去推敲这些数字。当加入虚构造函数时，使用其他编译器处理仅使该类占用的空间增加4个字节。因为我们不是编译器的编写者，所以无法得知编译器的秘密。然而可以确定的是，一个带有多重继承的派生对象必须表现出它好像有多个VPTR，它的每个含有虚函数的直接基类都有一个。就是那么简单。编译器的作者不论发明什么样的最优化技术，但是这些编译器必须产生相同的行为。

在前面的代码中，最奇怪的事情是**Bottom**构造函数中对**Top**的初始化程序。正常的情况下，不必担心直接基类以外的子对象的初始化，因为所有的类只照料它们的直接基类的初始化。然而，由于从**Bottom**到**Top**有多个继承路径，因此依赖于中间类**Left**和**Right**将必需的初始化数据传递给基类导致了二义性——谁负责进行基类的初始化呢？基于这个原因，最高层派生类（most derived class）必须初始化一个虚基类。但是也对**Top**进行初始化的**Left**和**Right**构造函数中的表达式应该如何编写呢？当创建独立的**Left**或**Right**对象时，这些初始化表达式确实是必需的，但是当创建**Bottom**对象时，它们必须被忽略（因此，在**Bottom**构造函数的初

① 这些指针的出现说明为什么**B**的长度远大于4个整型变量的长度。这是虚基类的部分开销。还有VPTR的开销，这归因于虚析构函数。

② 再说明一次，基类必须有虚析构函数，但是大部分编译器都能使这个例子编译通过。

始化程序中，这些表达式都为零——当**Left**和**Right**的构造函数在**Bottom**对象的语境中执行时，这些位置上的任何值都将被忽略)。编译器为程序员处理所有这一切，但是了解责任所在还是很重要的。必须始终保证，多重继承层次结构中的所有具体的（非抽象的）类都知道任何虚基类并对它们进行相应的初始化。

这些责任规则不仅仅适用于类的初始化，而且适用于所有跨越类继承层次结构的操作。现在考虑前面代码中的流插入符。使数据成为保护的，这样就可以“骗取”和访问 **operator<<(ostream&, const Bottom&)** 中继承来的数据。通常将打印各子对象的工作分配到其各个相应的类来进行，并且在需要的时候让派生类调用它的基类函数，这样做更有意义。就像下面的代码的说明，如果在程序中尝试使用 **operator<<()**，将会出现什么情况？

```

//: C09:VirtualBase2.cpp
// How NOT to implement operator<<.
#include <iostream>
using namespace std;

class Top {
    int x;
public:
    Top(int n) { x = n; }
    virtual ~Top() {}
    friend ostream& operator<<(ostream& os, const Top& t) {
        return os << t.x;
    }
};

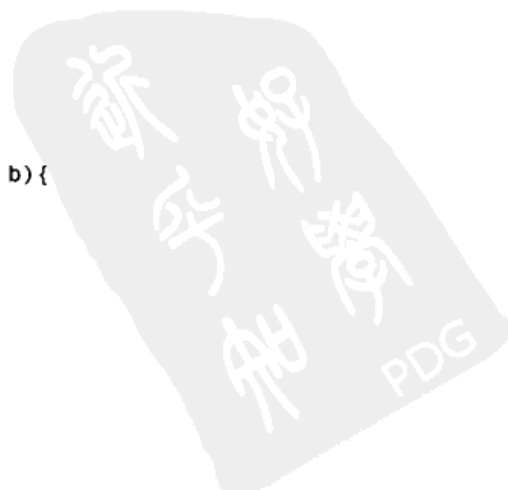
class Left : virtual public Top {
    int y;
public:
    Left(int m, int n) : Top(m) { y = n; }
    friend ostream& operator<<(ostream& os, const Left& l) {
        return os << static_cast<const Top&>(l) << ',' << l.y;
    }
};

class Right : virtual public Top {
    int z;
public:
    Right(int m, int n) : Top(m) { z = n; }
    friend ostream& operator<<(ostream& os, const Right& r) {
        return os << static_cast<const Top&>(r) << ',' << r.z;
    }
};

class Bottom : public Left, public Right {
    int w;
public:
    Bottom(int i, int j, int k, int m)
        : Top(i), Left(0, j), Right(0, k) { w = m; }
    friend ostream& operator<<(ostream& os, const Bottom& b){
        return os << static_cast<const Left&>(b)
            << ',' << static_cast<const Right&>(b)
            << ',' << b.w;
    }
};

int main() {
    Bottom b(1, 2, 3, 4);
    cout << b << endl; // 1,2,1,3,4
}
//:~

```



在通常处理方式中不能盲目地向上分摊责任，因为**Left**和**Right**每个流插入程序都调用了**Top**流插入程序，并且再出现数据的副本。另外，这里需要模仿编译器的初始化办法。一种解决办法是在类中提供特殊的函数，这种函数知道有关虚基类的情况，在打印输出的时候忽略虚基类（而将工作留给最高层派生类）：

```
//: C09:VirtualBase3.cpp
// A correct stream inserter.
#include <iostream>
using namespace std;

class Top {
    int x;
public:
    Top(int n) { x = n; }
    virtual ~Top() {}
    friend ostream& operator<<(ostream& os, const Top& t) {
        return os << t.x;
    }
};

class Left : virtual public Top {
    int y;
protected:
    void specialPrint(ostream& os) const {
        // Only print Left's part
        os << ',' << y;
    }
public:
    Left(int m, int n) : Top(m) { y = n; }
    friend ostream& operator<<(ostream& os, const Left& l) {
        return os << static_cast<const Top&>(l) << ',' << l.y;
    }
};

class Right : virtual public Top {
    int z;
protected:
    void specialPrint(ostream& os) const {
        // Only print Right's part
        os << ',' << z;
    }
public:
    Right(int m, int n) : Top(m) { z = n; }
    friend ostream& operator<<(ostream& os, const Right& r) {
        return os << static_cast<const Top&>(r) << ',' << r.z;
    }
};

class Bottom : public Left, public Right {
    int w;
public:
    Bottom(int i, int j, int k, int m)
        : Top(i), Left(0, j), Right(0, k) { w = m; }
    friend ostream& operator<<(ostream& os, const Bottom& b) {
        os << static_cast<const Top&>(b);
        b.Left::specialPrint(os);
        b.Right::specialPrint(os);
        return os << ',' << b.w;
    }
};
```

```
int main() {
    Bottom b(1, 2, 3, 4);
    cout << b << endl; // 1,2,3,4
} ///:~
```

**specialPrint()** 函数是 **protected** 的，因为它们只能被 **Bottom** 调用。**specialPrint()** 函数只输出自己的数据并忽略 **Top** 子对象，因为当这些函数被调用时，**Bottom** 插入程序将获得控制权。像 **Bottom** 的构造函数一样，**Bottom** 插入程序也必须知道虚基类。同样的推理适用于具有虚基类的层次结构中的赋值操作符，也适用想要分担层次结构中所有类的工作的任何成员函数或非成员函数。

在讨论了虚基类后，现在可以举例说明对象初始化的“全部情节”。因为虚基类引起共享子对象，共享发生之前它们就应该存在才有意义。所以子对象的初始化顺序遵循如下的规则递归地进行：

- 1) 所有虚基类子对象，按照它们在类定义中出现的位置，从上到下、从左到右初始化。
- 2) 然后非虚基类按通常顺序初始化。
- 3) 所有的成员对象按声明的顺序初始化。
- 4) 完整的对象的构造函数执行。

下面的程序举例说明了这个过程：

```
///: C09:VirtInit.cpp
// Illustrates initialization order with virtual bases.
#include <iostream>
#include <string>
using namespace std;

class M {
public:
    M(const string& s) { cout << "M " << s << endl; }
};

class A {
    M m;
public:
    A(const string& s) : m("in A") {
        cout << "A " << s << endl;
    }
    virtual ~A() {}
};

class B {
    M m;
public:
    B(const string& s) : m("in B") {
        cout << "B " << s << endl;
    }
    virtual ~B() {}
};

class C {
    M m;
public:
    C(const string& s) : m("in C") {
        cout << "C " << s << endl;
    }
    virtual ~C() {}
};

class D {
```



```

    M m;
public:
    D(const string& s) : m("in D") {
        cout << "D " << s << endl;
    }
    virtual ~D() {}
};

class E : public A, virtual public B, virtual public C {
    M m;
public:
    E(const string& s) : A("from E"), B("from E"),
        C("from E"), m("in E") {
        cout << "E " << s << endl;
    }
};

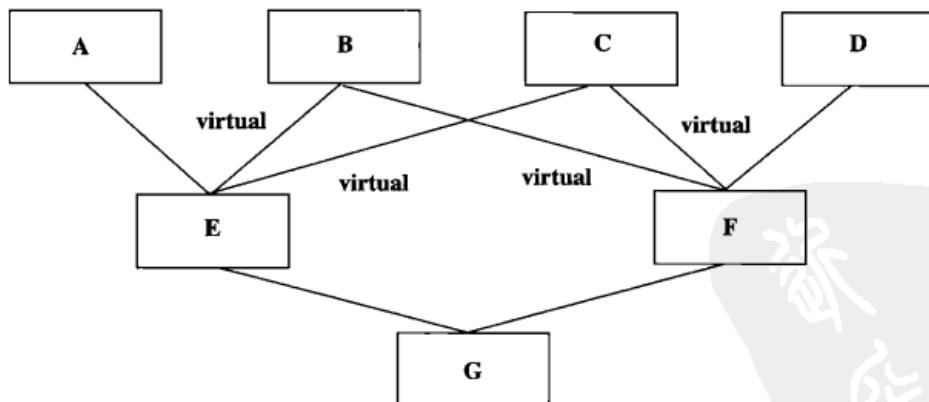
class F : virtual public B, virtual public C, public D {
    M m;
public:
    F(const string& s) : B("from F"), C("from F"),
        D("from F"), m("in F") {
        cout << "F " << s << endl;
    }
};

class G : public E, public F {
    M m;
public:
    G(const string& s) : B("from G"), C("from G"),
        E("from G"), F("from G"), m("in G") {
        cout << "G " << s << endl;
    }
};

int main() {
    G g("from main");
} ///:~

```

这段代码中的类可以用下图表示：



每一个类都有一个嵌入的**M**类型的成员。注意，只有4个派生类是虚拟的：**E**派生自**B**和**C**、**F**派生自**B**和**C**。这个程序的输出结果是：

```

M in B
B from G
M in C
C from G

```

```

M in A
A from E
M in E
E from G
M in D
D from F
M in F
F from G
M in G
G from main

```

**g**的初始化需要首先初始化它的**E**和**F**部分，但是**B**和**C**子对象首先被初始化，因为它们都是虚基类，并且二者的初始化在**G**的构造函数的初始化程序中进行，**G**是最高层派生类。类**B**没有基类，所以根据第3条规则，它的成员对象**m**被初始化，然后它的构造函数打印输出“**B** from **G**”，对于**E**的**C**子对象处理相同。**E**子对象的初始化需要先对**A**、**B**和**C**子对象进行初始化。因为**B**和**C**已经被初始化，于是**E**子对象的**A**子对象接着被初始化，然后是**E**子对象自己初始化。相同的情况重复出现在**g**的**F**子对象上，但是虚基类的初始化不重复进行。

## 9.6 名字查找问题

我们已经以子对象举例说明的二义性适用于任何名字，包括函数名。如果一个类有多个直接基类，就可以共享这些基类中那些同名的成员函数，如果要调用这些成员函数中的一个，那么编译器将不知道调用它们之中的哪一个。下面的程序举例将会报告这样一个错误：

```

//: C09:AmbiguousName.cpp {-xo}

class Top {
public:
    virtual ~Top() {}
};

class Left : virtual public Top {
public:
    void f() {}
};

class Right : virtual public Top {
public:
    void f() {}
};

class Bottom : public Left, public Right {};

int main() {
    Bottom b;
    b.f(); // Error here
} ///:~

```

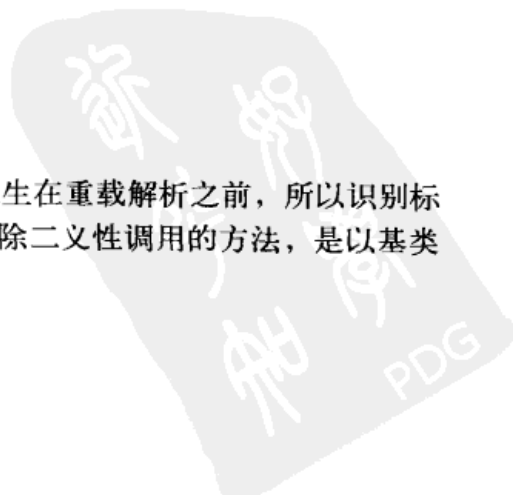
类**Bottom**已经继承了两个同名的函数（因为名字查寻发生在重载解析之前，所以识别标志是不恰当的），并且没有方法在它们之间进行选择。通常消除二义性调用的方法，是以基类名来限定函数的调用：

```

//: C09:BreakTie.cpp

class Top {
public:
    virtual ~Top() {}
};

```



```

class Left : virtual public Top {
public:
    void f() {}
};

class Right : virtual public Top {
public:
    void f() {}
};

class Bottom : public Left, public Right {
public:
    using Left::f;
};

int main() {
    Bottom b;
    b.f(); // Calls Left::f()
} ///:~

```

现在在**Bottom**的作用域中可以找到名字**Left::f**，所以完全不用考虑名字**Right::f**的查找问题。为了介绍**Left::f()**函数所能提供的更多额外功能，需要实现调用函数**Left::f()**的**Bottom::f()**函数。

在一个层次结构中的不同分支上存在的同名函数常常发生冲突。下面的继承层次结构不存在这样的问题：

```

//: C09:Dominance.cpp

class Top {
public:
    virtual ~Top() {}
    virtual void f() {}
};

class Left : virtual public Top {
public:
    void f() {}
};

class Right : virtual public Top {};

class Bottom : public Left, public Right {};

int main() {
    Bottom b;
    b.f(); // Calls Left::f()
} ///:~

```

程序在这里没有显式调用**Right::f()**。因为**Left::f()**是位于层次结构的最高层派生类，所以对**b.f()**语句的执行将调用**Left::f()**。为什么呢？现在假设**Right**不存在，这样就成为一个单一层次结构**Top <= Left <= Bottom**。在这里可以确定地预期由表达式**b.f()**调用的函数是**Left::f()**，因为一般的作用域规则是：一个派生类被认为嵌套在基类的作用域之内。一般情况下，如果类**A**直接或间接派生自类**B**，或换句话说，在继承层次结构中类**A**比类**B**处于“更高的派生层次”，<sup>①</sup>那么名字**A::f**就比名字**B::f**占优势（dominate）。因此，在同名的两个函数之间进

① 注意，对这个例子来说虚拟继承是至关重要的。如果**Top**不是虚基类，将存在多个虚**Top**子对象，并且二义性还将存在。多重继承的优越性只与虚基类一同存在。

行选择时，编译器将选择占优势的那个函数。如果没有占优势的名字，就会产生二义性。

下面的程序更进一步地举例说明了占优势的原则：

```

//: C09:Dominance2.cpp
#include <iostream>
using namespace std;

class A {
public:
    virtual ~A() {}
    virtual void f() { cout << "A::f\n"; }
};

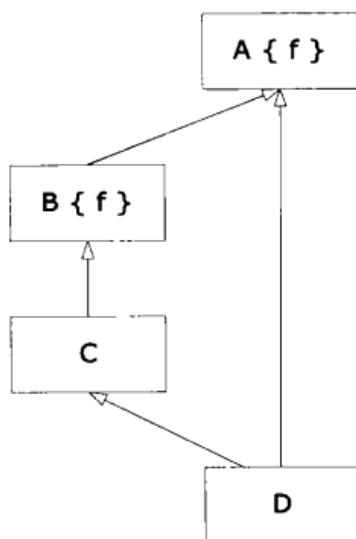
class B : virtual public A {
public:
    void f() { cout << "B::f\n"; }
};

class C : public B {};
class D : public C, virtual public A {};

int main() {
    B* p = new D;
    p->f(); // Calls B::f()
    delete p;
} ///:~

```

这个层次结构的类图如下：



类A是类B的基类（在这个例子中是直接基类），所以名字B::f比名字A::f占优势。

## 9.7 避免使用多重继承

当提到关于是否使用多重继承的问题时，至少要回答如下两个问题：

- 1) 是否需要通过新类来显示两个类的公共接口？（换句话说，如果一个类能够包含在另一个类中，那么仅有它的某些接口暴露在一个新类中。）
- 2) 需要向上类型转换成为两个基类类型吗？（当基类的数量多于两个时也适用。）

如果可以对上面任何一个问题回答“不是”，那么就可以避免使用MI，并且应该这样做。

请看这样的情况，一个类只是作为一个函数的参数需要向上进行类型转换。在这种情况下



下，这个类就可以被嵌入，并且在新类中有一个自动类型转换函数提供产生一个指向嵌入的对象的引用。任何时候，如果要将新类的一个对象作为参数传递给某个期盼嵌入对象的函数，这就需要使用类型转换函数。<sup>①</sup>然而，类型转换不能用于普通的多态成员函数的选择；那需要使用继承机制来完成。推荐使用组合而不使用继承，从总体上来说这是个不错的设计指导原则。

## 9.8 扩充一个接口

多重继承最好的应用之一，涉及由第3方提供的脱离了程序员控制的代码。假设获得了这样一个库，它由一个头文件和一些编译好的成员函数组成，但没有这些成员函数的源代码。这个库是一个带有虚函数的类层次结构，并且包含一些能将指针指向类库中基类的全局函数；就是说，它使用了这些库对象的多态性。现在，假设使用这个库创建一个应用程序并利用基类的多态性编写了程序员自己的代码。

在软件项目开发的后期或在其维护期间，程序员可能发现由软件供应商提供的基类的接口并没有提供所需要的功能：提供的函数可能是非虚函数，但现在却需要它是个虚函数，或者接口中的虚函数完全地失效了，而该虚函数对于问题的解决却是至关重要的。使用多重继承可以解决这个问题。

例如，这里就是你得到的库的一个头文件：

```
//: C09:Vendor.h
// Vendor-supplied class header
// You only get this & the compiled Vendor.obj.
#ifndef VENDOR_H
#define VENDOR_H

class Vendor {
public:
    virtual void v() const;
    void f() const; // Might want this to be virtual...
    ~Vendor(); // Oops! Not virtual!
};

class Vendor1 : public Vendor {
public:
    void v() const;
    void f() const;
    ~Vendor1();
};

void A(const Vendor&);
void B(const Vendor&);
// Etc.
#endif // VENDOR_H ///:~
```

假设这个库很大，由多个派生类和一个较大的接口组成。注意，它还包括了函数A()和B()，这些函数都有一个基类对象的引用，并且都能利用多态性对其进行处理。这里是库的实现文件：

```
//: C09:Vendor.cpp {0}
// Assume this is compiled and unavailable to you.
#include "Vendor.h"
```

① Jerry Schwarz，输入输出流（iostream）的作者，曾在个别场合表示如果他重新设计iostream的话，很可能从iostream的设计中去掉多重继承，而采用多重流缓冲区和转换运算符。

```

#include <iostream>
using namespace std;

void Vendor::v() const { cout << "Vendor::v()" << endl; }

void Vendor::f() const { cout << "Vendor::f()" << endl; }

Vendor::~Vendor() { cout << "~Vendor()" << endl; }

void Vendor1::v() const { cout << "Vendor1::v()" << endl; }

void Vendor1::f() const { cout << "Vendor1::f()" << endl; }

Vendor1::~Vendor1() { cout << "~Vendor1()" << endl; }

void A(const Vendor& v) {
    // ...
    v.v();
    v.f();
    // ...
}

void B(const Vendor& v) {
    // ...
    v.v();
    v.f();
    // ...
} ///:~

```

一般很难在用户自己的软件项目中获得这个源代码。而获得的只不过是像**Vendor.obj**或**Vendor.lib**（或与使用的系统相配的文件后缀）这样编译好的文件。

问题发生在对这个库的使用中。首先，析构函数不是虚函数。<sup>⑨</sup>另外，**f()**没有被设计为虚函数；在这里，假定库的创造者决定它不需要是虚函数。用户还可能发现，作为基类的接口缺少解决问题所必要的函数。还可以假设用户已经编写了利用现有接口的一些代码（更不用说函数**A()**和**B()**，它们已经超出了用户的控制），并且不想修改它。

为了补救这个问题，用户可以创建一个自己的类接口，并且采用多重继承方法产生一组新的派生类，这些新派生类派生自用户创建的类接口和已存在的类：

```

//: C09:Paste.cpp
//{L} Vendor
// Fixing a mess with MI.
#include <iostream>
#include "Vendor.h"
using namespace std;

class MyBase { // Repair Vendor interface
public:
    virtual void v() const = 0;
    virtual void f() const = 0;
    // New interface function:
    virtual void g() const = 0;
    virtual ~MyBase() { cout << "~MyBase()" << endl; }
};

class Paste1 : public MyBase, public Vendor1 {
public:

```

⑨ 人们已经在商品化的C++库中看到了这点，至少在一些早期的库中是这样。

```

void v() const {
    cout << "Pastel::v()" << endl;
    Vendor1::v();
}
void f() const {
    cout << "Pastel::f()" << endl;
    Vendor1::f();
}
void g() const { cout << "Pastel::g()" << endl; }
~Pastel() { cout << "~Pastel()" << endl; }
};

int main() {
    Pastel& p1p = *new Pastel;
    MyBase& mp = p1p; // Upcast
    cout << "calling f()" << endl;
    mp.f(); // Right behavior
    cout << "calling g()" << endl;
    mp.g(); // New behavior
    cout << "calling A(p1p)" << endl;
    A(p1p); // Same old behavior
    cout << "calling B(p1p)" << endl;
    B(p1p); // Same old behavior
    cout << "delete mp" << endl;
    // Deleting a reference to a heap object:
    delete &mp; // Right behavior
} ///:~

```

在**MyBase**（它没有使用MI）中，**f()**和析构函数现在都是虚函数，并且在接口中加入了新的虚函数**g()**。现在，原始库中的每个派生类都必须重新创建，并在新的接口中利用MI混合。函数**Pastel::v()**和**Pastel::f()**只需要调用原始基类中的成员函数的版本。但是现在，如果将派生类对象向上类型转换为**MyBase**，就像在**main()**中：

```
MyBase* mp = p1p; // Upcast
```

任何通过**mp**执行的函数调用都将是多态的，包括**delete**。同样地，新的接口函数**g()**也可以通过**mp**来调用。下面是程序的输出结果：

```

calling f()
Pastel::f()
Vendor1::f()
calling g()
Pastel::g()
calling A(p1p)
Pastel::v()
Vendor1::v()
Vendor::f()
calling B(p1p)
Pastel::v()
Vendor1::v()
Vendor::f()
delete mp
~Pastel()
~Vendor1()
~Vendor()
~MyBase()

```

原始的库函数**A()**和**B()**仍然能够照常工作（假设新函数**v()**调用了它的基类版本）。现在析构函数是**virtual**的，并且展现了正确的行为。

虽然这个例子有点儿混乱，但在实践中确实发生过，并且这个例子清晰地演示了哪里需要使用多重继承：必须能够将派生类向上类型转换为两个基类类型。

## 9.9 小结

C++中MI存在的一个原因是因为C++是一种混合语言，并且不能像Smalltalk和Java那样实现一个整体的类层次结构。而C++允许形成许多继承树，所以有时可能需要将来自两棵或多棵树的接口关联形成一个新类。

如果在类的层次结构中没有“菱形”的继承结构出现，MI将是相当简单的（虽然基类中完全相同的那些函数识别标志仍然必须解析）。如果有菱形继承结构出现，就需要通过引入虚基类来消除重复子对象。这不仅增加了混乱，而且使接下来的表达方式变得更加复杂和低效。

多重继承已经被称做“百分之90的goto语句”。<sup>①</sup>这种形容似乎是适当的，像避免使用goto语句那样在平常的编程中最好避免使用MI，但有时候它却很有用。它在C++中的地位是“次要的”，但却是C++的更高级特征，这一特征设计是用来解决特殊情况下出现的问题。如果读者发现自己经常使用了它，那么就需要检查一下使用它的原因。问一下自己，“是必需要向上类型转换成为所有的基类类型吗？”如果答案是否定的，假如嵌入的所有类的实例都不需要进行向上类型转换，那么编程工作将会变得更加简单。

## 9.10 练习

- 9-1 创建一个基类X，这个类包含具有一个int型参数的一个构造函数、一个返回类型为void的无参成员函数f()。现在从基类X派生出类Y和Z，并为它们各自创建一个具有一个int型参数的构造函数。然后，再从类Y和Z派生出类A。创建类A的一个对象，并且为这个对象调用f()。利用显式消除二义性的方法来解决这个问题。
- 9-2 以练习9-1的结果作为开始，创建一个指向基类X的名为px的指针，将前面创建的类A的对象地址赋值给px。利用虚基类解决这个问题。现在调整基类X，这样就不必再为X内部的A调用构造函数。
- 9-3 以练习9-2的结果作为开始，删除对f()使用显式消除二义性的方法，并且看看是否可以通过px调用f()。跟踪它看看哪个函数被调用了。解决这个问题，使得在类的继承层次结构中可以调用正确的函数。
- 9-4 与一个makeNoise()函数声明一起，构造一个Animal接口类。与savePersonFromFire()函数声明一起，构造一个SuperHero接口类。在这两个接口类中放置一个move()函数声明。（记住构造接口的方法是使用纯虚函数。）现在定义3个单独的类：SuperlativeMan、Amoeba（一个性情无常的超级英雄）和TarantulaWoman，当Amoeba和Tarantula Woman实现Animal和SuperHero接口的时候，SuperlativeMan实现SuperHero的接口。定义两个全局函数animalSound(Animal\*)和saveFromFire(SuperHero\*)。寻求用这两个函数能够通过每个接口调用相应对象的所有方法。
- 9-5 重复上面的练习，但是利用模板而不是继承来实现接口，就像在Interfaces2.cpp中做的那样。
- 9-6 定义若干代表超级英雄（superhero）能力的具体的混入类（例如StopTrain、

① Zack Urlocker创造的一个短语。

**BendSteel**和**ClimbBuilding**等)。重新完成练习9-4, 从这些混入类中派生出**SuperHero**派生类, 并且调用它们的成员函数。

- 9-7 利用模板重复上面的练习, 用强大的超级英雄 (**superhero**) 混入类作为模板参数。利用模板强大的功能更好地为其他类服务。
- 9-8 从练习9-4中撤销**Animal**接口, 重新定义**Amoeba**使其仅实现**SuperHero**。现在定义一个从两个类**SuperlativeMan**和**Amoeba**继承来的**SuperlativeAmoeba**类。试着将**SuperlativeAmoeba**对象作为参数传递给**saveFromFire()**。为了让上面的调用正确进行, 需要做些什么? 如何使用虚继承来改变对象的长度?
- 9-9 继续上面的练习, 为练习9-4的**SuperHero**增加一个整型**strengthFactor**数据成员, 并在构造函数中对其进行初始化。在3个派生类中也加入构造函数来初始化**strengthFactor**。在**SuperlativeAmoeba**中, 必须要做哪些不同的工作?
- 9-10 继续上面的练习, 分别给两个类**SuperlativeMan**和**Amoeba** (但不包括**SuperlativeAmoeba**) 增加一个**eatFood()**成员函数, 这样两个版本的**eatFood()**函数获得不同的食物对象的类型 (所以这两个函数的识别标志不同)。在**SuperlativeAmoeba**中调用两个**eatFood()**函数中的任一个时必须做哪些工作? 为什么?
- 9-11 为**SuperlativeAmoeba**定义一个能够正确工作的输出流插入符和赋值操作符。
- 9-12 从层次结构中删除**SuperlativeAmoeba**, 并且修改**Amoeba**使它派生自两个类**SuperlativeMan** (它也是由**SuperHero**派生) 和**SuperHero**。在**SuperHero**和**SuperlativeMan** (采用完全相同的识别标志) 中实现一个虚函数**workout()**, 并且以**Amoeba**对象调用这个函数。哪个函数被调用了?
- 9-13 用组合而非继承重新定义**SuperlativeAmoeba**, 使它的行为类似于**SuperlativeMan**或**Amoeba**。利用转换运算符提供隐式向上类型转换。将这种方法和继承方法进行比较。
- 9-14 假设有一个预先编译好的**Person**类 (即只有头文件和编译好的目标文件)。假设**Person**还有一个非虚函数**work()**。通过从**Person**派生和使用**Person::work()**的一个实现, 让**SuperHero**能够成为一个行为适度且遵守规矩的普通的**Person**, 而让**SuperHero::work()**成为虚函数。
- 9-15 定义一个引用计数错误的日志混入类**ErrorLog**, 持有一个静态文件流, 可以用这个流发送消息。当引用计数大于零时该类打开流, 当引用计数归零时 (始终附加在文件上) 关闭流。让多个类的对象能够向静态日志流发送消息。通过**ErrorLog**中的跟踪语句, 观察流的打开和关闭。
- 9-16 修改**BreakTie.cpp**, 在其中加入派生 (非虚派生) 自**Bottom**的名为**VeryBottom**的类。除非在**using**中对**f**的声明将“左”变为“右”, **VeryBottom**应该看起来就和**Bottom**一样。修改**main()**函数, 实例化一个**VeryBottom**而非**Bottom**对象。请问, 将调用哪个**f()**?

## 设计模式

“……描述一个在我们周围一再出现的问题，然后描述解决这个问题的核心方法，这样就能够无数次地使用这个解决方法而不必重复劳动。”——**Christopher Alexander**  
本章介绍程序设计的重要和非传统的“模式”方法。

“设计模式”（design pattern）运动或许是在面向对象设计方法学前进过程中的最新、最重要的一步，最初把设计模式概念载入编年史的，是Gamma、Helm、Johnson和Vlissides等4人合编的《Design Patterns》（Addison Wesley, 1995）<sup>①</sup>一书，这本书一般也被称为“四人帮”（Gang of Four, GoF）书。“四人帮”针对问题的特定类型提出了23种解决方案。在本章中，讨论设计模式的基本概念并且给出一些代码示例，用于说明精选出来的设计模式。希望这样能够促使大家研读更多关于设计模式的资料，设计模式当今已经成为面向对象程序设计的几乎所有必须掌握的语汇的重要源泉<sup>②</sup>。

### 10.1 模式的概念

最初，可以将模式看做解决某一类特定问题的特别巧妙和具有洞察力的方法。它体现出一支开发团队从一个问题的所有角度出发做出全面的分析后，提出的最通用最灵活的对这类问题的解决方案。这类问题也许是读者以前曾经遇到和解决过的问题，但是读者那种解决方案大概没有即将看到的在模式中体现出的完整性。此外，模式的存在独立于任何特定的实现方法，我们可以用多种方法来实现它。

虽然称为“设计模式”，它们实际上与设计领域并无联系。模式与传统的关于分析、设计和实现的思想方法有所不同。模式体现了一个程序内部完整思想，因此它也能够跨越分析阶段和高层设计阶段。然而，因为模式常常有一个直接的代码实现，所以在底层设计或编码实现之前很难将其表示出来（在进入这些阶段之前，人们可能不会认识到需要某种特定的模式）。

可以把模式的基本概念看做一般情况下程序设计的基本概念：增加一些抽象层。当人们对某事物进行抽象的时候，隔离特定的细节，最直接的动机之一是为了使变化的事物与不变的事物分离开。做到这一点的另一个方法是，一旦发现程序中的某些部分可能被修改，那么就要阻止那些修改在代码中到处传播副作用。如果做到了这一点，代码不仅比较容易阅读和理解，而且也比较容易维护——这样做会带来一个注定的结果，那就是在软件开发的全过程中降低成本。

开发一种优雅和可维护的软件设计最困难的部分，常常是发现所谓“变化向量”（the vector of change）。（在这里，“向量”应理解为自然科学中的最大梯度，而不是一个容器类。）这就意味着寻找系统中变化的最重要的事物，换句话说，去寻找系统中开发成本最高的地方。一旦找到这个“变化向量”，就可以围绕这个焦点来构建系统的设计。

因此，设计模式的目标是封装变化（encapsulate change）。如果从这点来看，在本书中，

① 为方便起见，书中的例子都是使用C++描述的；遗憾的是这种标准出现在C++前的方言缺乏一些诸如STL容器等现代语言特征。

② 许多材料来源于“Thinking in Patterns: Problem-Solving Techniques using Java”，可从网站www.MindView.net上得到。

读者已经看到了一些设计模式。例如，可以把继承（inheritance）想象为一个设计模式（尽管是由编译器提供的一个实现）。它表示行为不同（这是变化的事物）的一些对象，它们具有相同的接口（这就是所谓不变的事物）。组合（composition）也可以被认为是一种模式，因为可以改变——动态或静态地改变——实现类的对象，因此而改变类的工作方式。然而正常情况下，由编程语言直接支持的特性不能被归类为设计模式。

读者也已经看到了“四人帮”书中出现的另外一个模式：迭代器（iterator）。在STL的设计中它被当做基本的工具来使用，这在本教材中较早时已经讨论过。当分步骤和依次挑选容器中的元素时，迭代器隐藏容器的特殊的实现。允许使用迭代器编写通用的代码，对某一范围内的所有元素进行操作，而不必关心保存这些元素的容器。因此，这些通用的代码可以和任何能够生成迭代器的容器一起使用。

### 组合优于继承

“四人帮”的最重要的贡献也许并不在于提出了模式的概念，而在于在该书的第1章中介绍的那句格言：“对象组合优于类继承。”理解继承和多态性如此具有挑战性，以至于人们可能对这些技术赋予了不适当的重要性。我们看到许多由于“继承嗜好”而导致的过于复杂的设计（包括我们自己的设计）——比如，由于坚持到处使用继承致使许多多重继承设计得到发展。

《极限编程》（Extreme programming）的指导原则之一是“只要能用，就做最简单的”。一个似乎需要继承的设计常常能够戏剧性地使用组合来代替而大大简化，从而使其更加灵活，在学习过本章中的一些设计模式之后读者将会理解这一点。因此，在考虑一个设计时，问问自己：“使用组合是不是更简单？这里真的需要继承吗？它能带来什么好处？”

## 10.2 模式分类

“四人帮”讨论了23个模式，按照下面3种目的分类（对所有模式都围绕可能变化的特定方面考虑）：

1) **创建型 (Creational)**：用于怎样创建一个对象。通常包括隔离对象创建的细节，这样代码不依赖于对象是什么类型，因此在增加一种新的对象类型时不需要改变代码。本章将介绍单件（Singleton）模式、工厂（Factory）模式和构建器（Builder）模式。

2) **结构型 (Structural)**：影响对象之间的连接方式，确保系统的变化不需要改变对象间的连接。结构型模式常常由工程项目限制条件来支配。本章中将看到代理（Proxy）模式和适配器（Adapter）模式。

3) **行为型 (Behavioral)**：在程序中处理具有特定操作类型的对象。这些对象封装要执行的操作过程，比如解释一种语言、实践一个请求、遍历一个序列（如像在一个迭代器内）或者实现一个算法。本章包含命令（Command）模式、模板方法（Template Method）模式、状态（State）模式、策略（Strategy）模式、职责链（Chain of Responsibility）模式、观察者（Observer）模式、多派遣（Multiple Dispatching）模式和访问者（Visitor）模式的例子。

“四人帮”对23个模式的每个模式都用一节篇幅进行讨论，给出一个或多个例子，这些例子通常用C++描述，有时也用Smalltalk描述。本书不重复在“四人帮”书中阐述的那些模式的细节，既然那本书自成体系，就应该单独学习。在此提供的描述和例子旨在给读者一个关于模式的大致理解，以便对模式是什么和模式的重要性有一个感性的认识。

### 特征、习语和模式

接下来的内容已经超出“四人帮”书中的范围。自从“四人帮”的书出版以来，出现了更

多的模式，对于定义设计模式有了更细致的过程。<sup>⑤</sup>这是重要的，因为识别新的模式或适当地描述它们是不容易的。例如，对于什么是设计模式，在流行的文献中有一些混乱的定义和描述。模式不是琐碎的，它们通常也不是由某种编程语言中内部特征来表现。例如，构造函数和析构函数可以被称为“保证初始化和清除的设计模式”。对面向对象编程来说，这些是重要的和必要的结构，但它们是常规语言的特征，还没有丰富到足以被看成设计模式的地步。

另外一个非模式的例子就是来自各种形式的聚合。聚合是面向对象编程中的一个完全基本的原则：用一些对象制造另一些对象。然而，有时这种办法被错误地归类为一种模式。这是遗憾的，因为它打乱了设计模式的思想，暗示人们将第1次看到且感到惊奇的任何事物都归结为一种设计模式。

Java语言提供了另外一个使人误解的例子：JavaBeans 规格说明的设计者决定把简单的“get/set”命名约定称为一种设计模式（比如，**getInfo()**返回一个**Info**属性，而**setInfo()**改变这个属性）。这只是一个普通的命名约定，而不能构成设计模式。

### 10.3 简化习语

在讨论更复杂的技术之前，看一些能够保持代码简明的基本方法是有帮助的。

#### 10.3.1 信使

信使（messenger）<sup>⑥</sup>是这些方法中最微不足道的一个，它将消息封装到一个对象中到处传递，而不是将消息的所有片段分开进行传递。注意，没有信使，下面例子中的**translate()**的代码读起来将非常缺乏条理：

```
//: C10:MessengerDemo.cpp
#include <iostream>
#include <string>
using namespace std;

class Point { // A messenger
public:
    int x, y, z; // Since it's just a carrier
    Point(int xi, int yi, int zi) : x(xi), y(yi), z(zi) {}
    Point(const Point& p) : x(p.x), y(p.y), z(p.z) {}
    Point& operator=(const Point& rhs) {
        x = rhs.x;
        y = rhs.y;
        z = rhs.z;
        return *this;
    }
    friend ostream&
    operator<<(ostream& os, const Point& p) {
        return os << "x=" << p.x << " y=" << p.y
            << " z=" << p.z;
    }
};

class Vector { // Mathematical vector
public:
    int magnitude, direction;
    Vector(int m, int d) : magnitude(m), direction(d) {}
};
```

⑤ 最新信息查询请登录<http://hillside.net/patterns>。

⑥ 这是Bill Venner取的名字，在其他地方有别的名称。



```
};

class Space {
public:
    static Point translate(Point p, Vector v) {
        // Copy-constructor prevents modifying the original.
        // A dummy calculation:
        p.x += v.magnitude + v.direction;
        p.y += v.magnitude + v.direction;
        p.z += v.magnitude + v.direction;
        return p;
    }
};

int main() {
    Point p1(1, 2, 3);
    Point p2 = Space::translate(p1, Vector(11, 47));
    cout << "p1: " << p1 << " p2: " << p2 << endl;
} ///:~
```

代码在这里做了简单化处理以防混乱。

既然信使的目标只是为了携带数据，可将这些数据安排为公有成员以便访问。然而，也有理由将这些数据设为私有成员。

### 10.3.2 收集参数

信使的大兄弟是收集参数 (Collecting Parameter)，它的工作就是从传递给它的函数中获取信息。通常，当收集参数被传递给多个函数的时候使用它，就像蜜蜂在采集花粉一样。

容器对于收集参数特别有用，因为它已经设置为动态增加对象：

```
///: C10:CollectingParameterDemo.cpp
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class CollectingParameter : public vector<string> {};

class Filler {
public:
    void f(CollectingParameter& cp) {
        cp.push_back("accumulating");
    }
    void g(CollectingParameter& cp) {
        cp.push_back("items");
    }
    void h(CollectingParameter& cp) {
        cp.push_back("as we go");
    }
};

int main() {
    Filler filler;
    CollectingParameter cp;
    filler.f(cp);
    filler.g(cp);
    filler.h(cp);
    vector<string>::iterator it = cp.begin();
    while(it != cp.end())
        cout << *it++ << " ";
    cout << endl;
} ///:~
```



收集参数必须有一些方法用来设置值或者插入值。注意，根据这个定义信息可以被当做收集参数来使用。问题的关键是收集参数通过接收它的函数进行传递和修改。

## 10.4 单件

单件 (Singleton) 也许是“四人帮”给出的最简单的设计模式，它是允许一个类有且仅有一个实例的方法。下面的程序显示在C++中如何实现一个单件模式：

```
//: C10:SingletonPattern.cpp
#include <iostream>
using namespace std;

class Singleton {
    static Singleton s;
    int i;
    Singleton(int x) : i(x) { }
    Singleton& operator=(Singleton&); // Disallowed
    Singleton(const Singleton&);      // Disallowed
public:
    static Singleton& instance() { return s; }
    int getValue() { return i; }
    void setValue(int x) { i = x; }
};

Singleton Singleton::s(47);

int main() {
    Singleton& s = Singleton::instance();
    cout << s.getValue() << endl;
    Singleton& s2 = Singleton::instance();
    s2.setValue(9);
    cout << s.getValue() << endl;
} ///:~
```

创建一个单件模式的关键是防止客户程序员获得任何控制其对象生存期的权利。为了做到这一点，声明所有的构造函数为私有，并且防止编译器隐式生成任何构造函数。注意，拷贝构造函数和赋值操作符（这两个方法都故意没有实现，因为它们根本就不会被调用）被声明为私有，以便防止任何这类复制的动作产生。

还必须决定如何去创建这个对象。在这里，它被静态创建的，但也可以等待，直到客户程序员提出要求再根据要求进行创建。这种方式称作惰性初始化 (lazy initialization)，这种做法，只在创建对象的代价不大，并且并不总是需要它的情况下才有意义。

如果返回的是一个指针而不是引用，用户可能会不小心删除此指针，因此上述实现被认为是最安全的（析构函数也可以声明为私有或者保护的，以便缓和此问题）。在任何情况下，对象应该私有保存。

通过公有成员函数来提供对其对象的访问。在这里，**instance()**产生**Singleton**对象的引用。其余的接口 (**getValue()**和**setValue()**) 是常见的类接口。

注意，这种方法并没有限制只创建一个对象。这种技术也支持创建有限个对象的对象池。然而在这种情况下，可能遇到池中共享对象的问题。如果这是一个问题，可以采取创建一个对共享对象进出对象池登记的方法来解决。

### 单件的变体

一个类中的任何**static**静态成员对象都表示一个单件：有且仅有一个对象被创建。因此，

从某种意义上讲，编程语言对单件技术提供了直接支持；我们自然是在常规基础上使用它。然而，对于**static**对象（类成员或者非类成员）来说有个问题：就是初始化的顺序的确定，如本书第1卷所述。如果一个静态对象依赖于另一个对象，那么将这些对象按正确的顺序进行初始化是很重要的。

在第1卷中，已经指出了如何在一个函数中定义一个静态对象来控制初始化顺序。这种方法延迟对象的初始化，直到在该函数第1次被调用时才进行初始化。如果该函数返回一个静态对象的引用，就可以达到单件的效果，这样就消除了可能由静态初始化引起的许多烦恼。例如，假如想在第1次调用某个函数时创建一个日志文件，该函数返回了那个日志文件的引用。下面这个头文件将完成这个任务：

```
//: C10:LogFile.h
#ifndef LOGFILE_H
#define LOGFILE_H
#include <fstream>
std::ofstream& logfile();
#endif // LOGFILE_H ///:~
```

函数的实现必须不是内联的（must not be inlined），因为那将意味着整个函数包括在其中定义的静态对象，在任何包含它的翻译单元中都被复制，这就违反了C++的一次定义（one-definition）规则<sup>①</sup>。这肯定阻碍试图控制初始化顺序的努力（但可能以微妙的、很难发现的形式出现）。因此函数的实现必须分开：

```
//: C10:LogFile.cpp {0}
#include "LogFile.h"
std::ofstream& logfile() {
    static std::ofstream log("Logfile.log");
    return log;
} ///:~
```

现在**log**对象不被初始化，直至函数**logfile()**第1次调用时才被初始化。因此，如果创建一个函数：

```
//: C10:UseLog1.h
#ifndef USELOG1_H
#define USELOG1_H
void f();
#endif // USELOG1_H ///:~
```

在函数的实现中使用**logfile()**：

```
//: C10:UseLog1.cpp {0}
#include "UseLog1.h"
#include "LogFile.h"
void f() {
    logfile() << __FILE__ << std::endl;
} ///:~
```

并且在另一个文件中再次使用**logfile()**：

```
//: C10:UseLog2.cpp
//{L} LogFile UseLog1
#include "UseLog1.h"
#include "LogFile.h"
using namespace std;
```

① C++ 标准要求：“任何翻译单元都不得对任何变量、函数、类类型、枚举类型或模板等多次定义。在程序中使用的非内联函数或对象只能定义一次。”

```

void g() {
    logfile() << __FILE__ << endl;
}

int main() {
    f();
    g();
} ///:~

```

直至首次调用函数**f()**时，对象**log**才被创建。

可以很容易地将在一个成员函数内部的静态对象的创建与单件类结合在一起。

**SingletonPattern.cpp**可用这个方法做如下修改：<sup>①</sup>

```

//: C10:SingletonPattern2.cpp
// Meyers' Singleton.
#include <iostream>
using namespace std;

class Singleton {
    int i;
    Singleton(int x) : i(x) { }
    void operator=(Singleton&);
    Singleton(const Singleton&);
public:
    static Singleton& instance() {
        static Singleton s(47);
        return s;
    }
    int getValue() { return i; }
    void setValue(int x) { i = x; }
};

int main() {
    Singleton& s = Singleton::instance();
    cout << s.getValue() << endl;
    Singleton& s2 = Singleton::instance();
    s2.setValue(9);
    cout << s.getValue() << endl;
} ///:~

```

如果两个单件彼此依赖，就会产生一个特别有趣的情况，如下所示：

```

//: C10:FunctionStaticSingleton.cpp

class Singleton1 {
    Singleton1() {}
public:
    static Singleton1& ref() {
        static Singleton1 single;
        return single;
    }
};

class Singleton2 {
    Singleton1& s1;
    Singleton2(Singleton1& s) : s1(s) {}
public:
    static Singleton2& ref() {
        static Singleton2 single(Singleton1::ref());
    }
};

```

① 这被称为Meyers单件，以它的创建者Scott Meyers命名。



```

        return single;
    }
    Singleton1& f() { return s1; }
};

int main() {
    Singleton1& s1 = Singleton2::ref().f();
} ///:~

```

当调用**Singleton2::ref()**时，它导致其惟一的**Singleton2**对象被创建。在这个对象的创建过程中，**Singleton1::ref()**被调用，这导致其惟一的**Singleton1**对象被创建。因为这种技术不依赖连接或装载的顺序，因此程序员能够很好地控制初始化的全过程，而导致较少的错误。

单件的另外一种变体采用将一个对象的“单件属性 (Singleton-ness)”从其实现中分离出来的方法。使用第5章提到的“奇特的递归模板模式 (Curiously Recurring Template Pattern)”来实现：

```

///: C10:CuriousSingleton.cpp
// Separates a class from its Singleton-ness (almost).
#include <iostream>
using namespace std;
template<class T> class Singleton {
    Singleton(const Singleton&);
    Singleton& operator=(const Singleton&);
protected:
    Singleton() {}
    virtual ~Singleton() {}
public:
    static T& instance() {
        static T theInstance;
        return theInstance;
    }
};

// A sample class to be made into a Singleton
class MyClass : public Singleton<MyClass> {
    int x;
protected:
    friend class Singleton<MyClass>;
    MyClass() { x = 0; }
public:
    void setValue(int n) { x = n; }
    int getValue() const { return x; }
};

int main() {
    MyClass& m = MyClass::instance();
    cout << m.getValue() << endl;
    m.setValue(1);
    cout << m.getValue() << endl;
} ///:~

```

**MyClass**通过下面3个步骤产生一个单件：

- 1) 声明其构造函数为私有或保护的。
- 2) 声明类**Singleton<MyClass>**为友元。
- 3) 从**Singleton<MyClass>**派生出**MyClass**。

在第3步中的自引用可能令人难以置信，然而正如第5章所述，因为这只是对模板**Singleton**中模板参数的静态依赖。换句话说，类**Singleton<MyClass>**的代码之所以能够被编译器实例化，是因为它不依赖于类**MyClass**的大小。只是在后来，当函数**Singleton<**

**MyClass>:: instance( )**第1次被调用时,才需要类**MyClass**的大小,而此时编译器已经知道类**MyClass**的大小。<sup>①</sup>

有趣的是,像单件这样简单的设计模式能有多么复杂,这里实际上还没有涉及线程安全的问题。最后说明一点,单件应该少用。真正的单件对象很少出现,而最终单件应该用于代替全局变量。<sup>②</sup>

## 10.5 命令：选择操作

命令 (command) 模式的结构很简单,但是对于消除代码间的耦合 (decoupling) ——清理代码——却有着重要的影响。

在《Advanced C++: Programming Styles And Idioms》(Addison Wesley, 1992) 一书中, Jim Coplien 创造了术语函子 (functor), 它表示一个对象, 该对象的惟一目的是封装一个函数 (由于“函子”在数学上有其特定的意义, 这里将用更加明确的术语函数对象 (function object) 来代替它)。其特点就是消除被调用函数的选择与那个函数被调用的位置之间的联系。

GoF书中也提到这个术语,但是没有使用。然而,函数对象的话题却在那本书的很多模式中被反复论及。

从最直观的角度来看,命令模式就是一个函数对象:一个作为对象的函数。通过将函数封装为对象,就能够以参数的形式将其传递给其他函数或者对象,告诉它们在履行请求的过程中执行特定的操作。可以说,命令模式是携带行为信息的信使。

```
//: C10:CommandPattern.cpp
#include <iostream>
#include <vector>
using namespace std;

class Command {
public:
    virtual void execute() = 0;
};

class Hello : public Command {
public:
    void execute() { cout << "Hello "; }
};

class World : public Command {
public:
    void execute() { cout << "World! "; }
};

class IAm : public Command {
public:
    void execute() { cout << "I'm the command pattern!"; }
};

// An object that holds commands:
class Macro {
    vector<Command*> commands;
```

① 在《Modern C++ Design》一书中, Andrei Alexandrescu提出了一种优越的基于策略的解决方案实现单件模式。

② 参看Hyslop 和Sutter 发表在2003年3月的《issue of CUJ》上的文章“Once is Not Enough”可以了解更详细的信息。

```

public:
    void add(Command* c) { commands.push_back(c); }
    void run() {
        vector<Command*>::iterator it = commands.begin();
        while(it != commands.end())
            (*it++)->execute();
    }
};

int main() {
    Macro macro;
    macro.add(new Hello);
    macro.add(new World);
    macro.add(new IAm);
    macro.run();
} ///:~

```

命令模式的主要特点是允许向一个函数或者对象传递一个想要的动作。上述例子提供了将一系列需要一起执行的动作集进行排队的方法。在这里，可以动态创建新的行为，某些事情通常只能通过编写新的代码来完成，而在上述例子中可以通过解释一个脚本来实现（如果需要实现的东西很复杂请参考解释器模式）。

GoF认为“命令模式是回调（callback）的面向对象的替代物”，然而这里的单词“back”是回调概念的重要的一部分——回调返回到回调的创建者所在的位置。另一方面，对于一个命令对象来说，典型的做法仅仅是创建它并且将之传递给一些函数或者对象，而不是自始至终以其他方式联系命令对象。

命令模式的一个常见的例子就是在应用程序中“撤销（undo）操作”功能的实现。每次在用户进行某项操作的时候，相应的“撤销操作”命令对象就被置入一个队列中。而每个命令对象被执行后，程序的状态就倒退一步。

### 利用命令模式消除与事件处理的耦合

正如读者将在下一章中要看到的，采用并发（concurrency）技术的原因之一是为了更容易地掌握事件驱动编程（event-driven programming），在事件驱动方式的编程中，这些事件出现的地方是不可预料的。例如，当程序正在执行一个操作时，用户按下“退出”按钮并且希望程序能够快速响应。

使用并发的论据是它能够防止程序中代码段间的耦合。也就是说，如果运行一个独立的线程用于监视退出按钮，程序的“正常”操作无须知道有关退出按钮或者其他需要监视的操作。

然而，一旦读者理解耦合是一个问题，就可以用命令模式来避免它。每个“正常”的操作必须周期性地调用一个函数来检查事件的状态，而通过命令模式，这些“正常”操作不需要知道有关它们所检查的事件的任何信息，也就是说它们已经与事件处理代码分离开来。

```

//: C10:MulticastCommand.cpp {RunByHand}
// Decoupling event management with the Command pattern.
#include <iostream>
#include <vector>
#include <string>
#include <ctime>
#include <cstdlib>
using namespace std;

// Framework for running tasks:

```

```

class Task {
public:
    virtual void operation() = 0;
};

class TaskRunner {
    static vector<Task*> tasks;
    TaskRunner() {} // Make it a Singleton
    TaskRunner& operator=(TaskRunner&); // Disallowed
    TaskRunner(const TaskRunner&); // Disallowed
    static TaskRunner tr;
public:
    static void add(Task& t) { tasks.push_back(&t); }
    static void run() {
        vector<Task*>::iterator it = tasks.begin();
        while(it != tasks.end())
            (*it++)->operation();
    }
};

TaskRunner TaskRunner::tr;
vector<Task*> TaskRunner::tasks;

class EventSimulator {
    clock_t creation;
    clock_t delay;
public:
    EventSimulator() : creation(clock()) {
        delay = CLOCKS_PER_SEC/4 * (rand() % 20 + 1);
        cout << "delay = " << delay << endl;
    }
    bool fired() {
        return clock() > creation + delay;
    }
};

// Something that can produce asynchronous events:
class Button {
    bool pressed;
    string id;
    EventSimulator e; // For demonstration
public:
    Button(string name) : pressed(false), id(name) {}
    void press() { pressed = true; }
    bool isPressed() {
        if(e.fired()) press(); // Simulate the event
        return pressed;
    }
    friend ostream&
    operator<<(ostream& os, const Button& b) {
        return os << b.id;
    }
};

// The Command object
class CheckButton : public Task {
    Button& button;
    bool handled;
public:
    CheckButton(Button & b) : button(b), handled(false) {}
    void operation() {
        if(button.isPressed() && !handled) {
            cout << button << " pressed" << endl;

```





```

        handled = true;
    }
}
};

// The procedures that perform the main processing. These
// need to be occasionally "interrupted" in order to
// check the state of the buttons or other events:
void procedure1() {
    // Perform procedure1 operations here.
    // ...
    TaskRunner::run(); // Check all events
}

void procedure2() {
    // Perform procedure2 operations here.
    // ...
    TaskRunner::run(); // Check all events
}

void procedure3() {
    // Perform procedure3 operations here.
    // ...
    TaskRunner::run(); // Check all events
}

int main() {
    srand(time(0)); // Randomize
    Button b1("Button 1"), b2("Button 2"), b3("Button 3");
    CheckButton cb1(b1), cb2(b2), cb3(b3);
    TaskRunner::add(cb1);
    TaskRunner::add(cb2);
    TaskRunner::add(cb3);
    cout << "Control-C to exit" << endl;
    while(true) {
        procedure1();
        procedure2();
        procedure3();
    }
} //::~~

```

在这里，命令对象由被单件**TaskRunner**执行的**Task**表示。**EventSimulator**创建一个随机延迟时间，所以当周期性的调用函数**fired()**时，在某个随机时间段，其返回结果从**true**到**false**变化。**EventSimulator**对象在类**Button**中使用，模拟在某个不可预知的时间段用户事件发生的动作。**CheckButton**是**Task**的实现，在程序中通过所有“正常”代码对其进行周期性的检查——可以看到这些检查发生在函数**Procedure1()**、**Procedure2()**和**Procedure3()**的末尾。

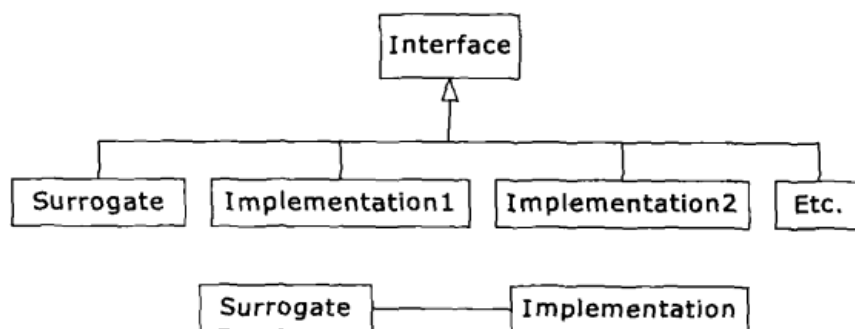
尽管这需要颇费点脑筋来设立命令对象，但是读者将在第11章中看到，如果采用线程处理方法则需要更多的考虑，小心预防并行编程中与生俱来的各种的困难问题，所以这种较简便的解决方法更可取。将**TaskRunner::run()**调用植入一个多线程处理的“计时器”对象中，也可以创建一个很简单的线程处理方案。这样做，可以消除所有“正常操作”（上述例子中的过程）与事件代码间的耦合。

## 10.6 消除对象耦合

代理（Proxy）模式和状态（State）模式都提供一个代理（Surrogate）类。代码与代理类打交道，而做实际工作的类隐藏在代理类背后。当调用代理类中的一个函数时，代理类仅转而

去调用实现类中相应的函数。这两种模式是如此相似，从结构上看，可以认为代理模式只是状态模式的一个特例。设想将这两者合理地混合在一起组成一个称为代理（Surrogate）设计模式，这肯定是一个很具有诱惑力的想法，但是这两个模式的内涵（intent）是不一样的。这样做很容易陷入“如果结构相同模式就相同”的思想误区。必须始终关注模式的内涵，从而明确它的功能到底是什么。

基本思想很简单：代理（Surrogate）类派生自一个基类，由平行地派生自同一个基类的一个或多个类提供实际的实现：



当一个代理对象被创建的时候，一个实现对象就分配给了它，代理对象就将函数调用发给实现对象。

从结构上来看，代理模式和状态模式的区别很简单：代理模式只有一个实现类，而状态模式有多个（一个以上）实现。（在GoF中）认为这两种设计模式的应用也不同：代理模式控制对其实现类的访问，而状态模式动态地改变其实现类。然而，如果广义理解“控制对实现类的访问”，则这两个模式似乎是一个连续体的两部分。

### 10.6.1 代理模式：作为其他对象的前端

如果按照上面的图结构实现代理模式，其实现代码如下：

```

//: C10:ProxyDemo.cpp
// Simple demonstration of the Proxy pattern.
#include <iostream>
using namespace std;
class ProxyBase {
public:
    virtual void f() = 0;
    virtual void g() = 0;
    virtual void h() = 0;
    virtual ~ProxyBase() {}
};

class Implementation : public ProxyBase {
public:
    void f() { cout << "Implementation.f()" << endl; }
    void g() { cout << "Implementation.g()" << endl; }
    void h() { cout << "Implementation.h()" << endl; }
};

class Proxy : public ProxyBase {
    ProxyBase* implementation;
public:
    Proxy() { implementation = new Implementation(); }
    ~Proxy() { delete implementation; }
    // Forward calls to the implementation:
    void f() { implementation->f(); }
}
  
```



```

    void g() { implementation->g(); }
    void h() { implementation->h(); }
};

int main() {
    Proxy p;
    p.f();
    p.g();
    p.h();
} ///:~

```

在某些情况下，类**Implementation**并不需要与类**Proxy**具有相同的接口——**Proxy**类可以任意“订购”（关联）**Implementation**类并且将函数调用提交给它，这就符合了代理的基本思想（值得注意的是，这种描述和GoF关于代理的定义不一致）。然而，使用共同的接口可以将代理的替代物插入客户代码中——编写客户代码只用来与原对象进行通信，不需对其进行修改以接受代理（这大概是使用代理的关键问题）。此外，通过共同的接口，**Implementation**被迫实现**Proxy**需要调用的所有函数。

代理模式与状态模式之间的不同之处在于它们所解决的问题不同。GoF中给出了代理模式的一般用途，描述如下：

1) **远程代理 (Remote proxy)**。为不同地址空间的对象提供代理。通过某些远程对象技术实现。

2) **虚拟代理 (Virtual proxy)**。根据需要提供一种“惰性初始化”方式来创建高代价的对象。

3) **保护代理 (Protection proxy)**。当不愿意客户程序员拥有被代理对象的全部访问权限时，使用保护代理。

4) **巧妙引用 (Smart reference)**。当访问被代理的对象时，增加额外的活动。引用计数 (reference counting) 就是一个例子：它用来跟踪被代理的某个特定对象被引用的次数，以实现写入时复制 (copy-on-write) 并且防止对象起别名。<sup>①</sup> 一个更简单的例子就是对特定函数的调用进行计数。

### 10.6.2 状态模式：改变对象的行为

状态模式产生一个可以改变其类的对象，当发现在大多数或者所有函数中都存在有条件的代码时，这种模式很有用。和代理模式一样，状态模式通过一个前端对象来使用后端实现对象履行其职责。然而，在前端对象生存期期间，状态模式从一个实现对象到另一个实现对象进行切换，以实现对于相同的函数调用产生不同的行为。如果在决定函数该做什么之前在每个函数内部做很多测试，那么这种方法是对实现代码的一种很好的改进。举个例子，在青蛙王子童话中，青蛙王子依照其所处的状态而有不同的行为。现在可以通过测试一个**bool**变量来实现：

```

//: C10:KissingPrincess.cpp
#include <iostream>
using namespace std;

class Creature {
    bool isFrog;
public:
    Creature() : isFrog(true) {}
    void greet() {
        if(isFrog)

```

① 参阅《C++编程思想》第1卷以获得关于引用计数更详细的知识。

```

        cout << "Ribbet!" << endl;
    else
        cout << "Darling!" << endl;
    }
    void kiss() { isFrog = false; }
};

int main() {
    Creature creature;
    creature.greet();
    creature.kiss();
    creature.greet();
} ///:~

```

然而，**greet()**等任何其他所有函数在执行操作前都必须测试变量**isFrog**，这样就使代码变得笨拙至极，特别是在系统中加入额外的状态时情况会更加严重。通过将操作委派给状态对象，这种情况就可以改变，代码从而得到了简化。

```

///: C10:KissingPrincess2.cpp
// The State pattern.
#include <iostream>
#include <string>
using namespace std;

class Creature {
    class State {
    public:
        virtual string response() = 0;
    };
    class Frog : public State {
    public:
        string response() { return "Ribbet!"; }
    };
    class Prince : public State {
    public:
        string response() { return "Darling!"; }
    };
    State* state;
public:
    Creature() : state(new Frog()) {}
    void greet() {
        cout << state->response() << endl;
    }
    void kiss() {
        delete state;
        state = new Prince();
    }
};

int main() {
    Creature creature;
    creature.greet();
    creature.kiss();
    creature.greet();
} ///:~

```

在这里，将实现类设计为嵌套或者私有并不是必需的，但是如果能做到的话，就会创建出更加清晰的代码。

注意，对状态类的改变将会自动地在所有的代码中进行传播，而不需要编辑这些类来完成改变。

## 10.7 适配器模式

适配器 (Adapter) 模式接受一种类型并且提供一个对其他类型的接口。当给定一个库或者具有某一接口的一段代码, 同时还给定另外一个库或者与前面那段代码的基本思想相同的一段代码而只是表达方式不一致时, 适配器模式将十分有用。通过调整彼此的表达方式以适配彼此, 将会迅速产生解决方法。

假设有个产生斐波那契数列的发生器类, 如下所示:

```
//: C10:FibonacciGenerator.h
#ifndef FIBONACCIGENERATOR_H
#define FIBONACCIGENERATOR_H

class FibonacciGenerator {
    int n;
    int val[2];
public:
    FibonacciGenerator() : n(0) { val[0] = val[1] = 0; }
    int operator()() {
        int result = n > 2 ? val[0] + val[1] : n > 0 ? 1 : 0;
        ++n;
        val[0] = val[1];
        val[1] = result;
        return result;
    }
    int count() { return n; }
};
#endif // FIBONACCIGENERATOR_H ///:~
```

由于它是一个发生器, 可以调用 **operator()** 来使用它, 如下所示:

```
//: C10:FibonacciGeneratorTest.cpp
#include <iostream>
#include "FibonacciGenerator.h"
using namespace std;

int main() {
    FibonacciGenerator f;
    for(int i = 0; i < 20; i++)
        cout << f.count() << ": " << f() << endl;
} ///:~
```

也许读者希望利用这个发生器来执行 STL 数值算法操作。遗憾的是, STL 算法只能使用迭代器才能工作, 这就存在接口不匹配的问题。解决方法就是创建一个适配器, 它将接受 **FibonacciGenerator** 并产生一个供 STL 算法使用的迭代器。由于数值算法只要求一个输入迭代器, 该适配器模式相当地直观 (为了某种目的, 它产生了一个 STL 迭代器, 如下所示):

```
//: C10:FibonacciAdapter.cpp
// Adapting an interface to something you already have.
#include <iostream>
#include <numeric>
#include "FibonacciGenerator.h"
#include "../C06/PrintSequence.h"
using namespace std;

class FibonacciAdapter { // Produce an iterator
    FibonacciGenerator f;
    int length;
public:
```

```

FibonacciAdapter(int size) : length(size) {}
class iterator;
friend class iterator;
class iterator : public std::iterator<
    std::input_iterator_tag, FibonacciAdapter, ptrdiff_t> {
    FibonacciAdapter& ap;
public:
    typedef int value_type;
    iterator(FibonacciAdapter& a) : ap{a} {}
    bool operator==(const iterator&) const {
        return ap.f.count() == ap.length;
    }
    bool operator!=(const iterator& x) const {
        return !(*this == x);
    }
    int operator*() const { return ap.f(); }
    iterator& operator++() { return *this; }
    iterator operator++(int) { return *this; }
};
iterator begin() { return iterator(*this); }
iterator end() { return iterator(*this); }
};

int main() {
    const int SZ = 20;
    FibonacciAdapter a1(SZ);
    cout << "accumulate: "
        << accumulate(a1.begin(), a1.end(), 0) << endl;
    FibonacciAdapter a2(SZ), a3(SZ);
    cout << "inner product: "
        << inner_product(a2.begin(), a2.end(), a3.begin(), 0)
        << endl;
    FibonacciAdapter a4(SZ);
    int r1[SZ] = {0};
    int* end = partial_sum(a4.begin(), a4.end(), r1);
    print(r1, end, "partial_sum", " ");
    FibonacciAdapter a5(SZ);
    int r2[SZ] = {0};
    end = adjacent_difference(a5.begin(), a5.end(), r2);
    print(r2, end, "adjacent_difference", " ");
} ///:~

```

通过被告知斐波那契数列的长度来初始化**FibonacciAdapter**。当创建**iterator**时，它仅获得一个包含**FibonacciAdapter**的引用，这样它就能够访问**FibonacciGenerator**和**length**。注意，相等比较忽略了右边的值，因为惟一重要的问题是判断发生器是否达到其长度。此外，**operator++()**没有修改迭代器；改变**FibonacciAdapter**状态的惟一操作是调用发生器**FibonacciGenerator**中的函数**operator()**。我们在迭代器的这个极其简单的版本上是侥幸成功的，因为对输入迭代器的约束条件十分严格；特别是，在该序列中每个值只能读取一次。

在函数**main()**中，可以看到所有4类不同的数值算法同**FibonacciAdapter**一起成功地通过了测试。

## 10.8 模板方法模式

应用程序结构框架允许从一个或一组类中继承以便创建一个新的应用程序，重用现存类中几乎所有的代码，并且覆盖其中一个或多个函数以便自定义所需要的应用程序。应用程序结构框架中的一个基本的概念是模板方法（Template Method）模式，它很典型地隐藏在覆盖的下

方，通过调用基类的不同函数（这里覆盖了其中一些函数以创建应用程序）来驱动程序运行。

模板方法模式的一个重要特征是它的定义在基类中（有时作为一个私有成员函数）并且不能改动——模板方法模式就是“坚持相同的代码”。它调用其他基类函数（就是那些被覆盖的函数）以便完成其工作，但是客户程序员不必直接调用这些函数，如下所示：

```
//: C10:TemplateMethod.cpp
// Simple demonstration of Template Method.
#include <iostream>
using namespace std;

class ApplicationFramework {
protected:
    virtual void customize1() = 0;
    virtual void customize2() = 0;
public:
    void templateMethod() {
        for(int i = 0; i < 5; i++) {
            customize1();
            customize2();
        }
    }
};

// Create a new "application":
class MyApp : public ApplicationFramework {
protected:
    void customize1() { cout << "Hello "; }
    void customize2() { cout << "World!" << endl; }
};

int main() {
    MyApp app;
    app.templateMethod();
} ///:~
```

驱动应用程序运行的“引擎”是模板方法模式。在GUI（图形用户界面）应用程序中，这个“引擎”就是主要的事件环。客户程序员只需提供**customize1()**和**customize2()**的定义，便可以令“应用程序”运行。

## 10.9 策略模式：运行时选择算法

注意，模板方法模式是“坚持相同的代码”，而被覆盖的函数是“变化的代码”。然而，这种变化在编译时通过继承被固定下来。按照“组合优于继承”的格言，可以利用组合来解决将变化的代码从“坚持相同的代码”中分开的问题，从而产生策略（Strategy）模式。这种方法有一个明显的好处：在程序运行时，可以插入变化的代码。策略模式也加入了“语境”，它可以是一个代理类，这个类控制着对特定策略对象的选择和使用——就像状态模式一样。

“策略”的意思就是：可以使用多种方法来解决某个问题——即“条条大路通罗马”。现在考虑一下忘记了某个人姓名时的情境。这里的程序可以用不同方法解决这个问题：

```
//: C10:Strategy.cpp
// The Strategy design pattern.
#include <iostream>
using namespace std;

class NameStrategy {
public:
```

```

    virtual void greet() = 0;
};

class SayHi : public NameStrategy {
public:
    void greet() {
        cout << "Hi! How's it going?" << endl;
    }
};

class Ignore : public NameStrategy {
public:
    void greet() {
        cout << "(Pretend I don't see you)" << endl;
    }
};

class Admission : public NameStrategy {
public:
    void greet() {
        cout << "I'm sorry. I forgot your name." << endl;
    }
};

// The "Context" controls the strategy:
class Context {
    NameStrategy& strategy;
public:
    Context(NameStrategy& strat) : strategy(strat) {}
    void greet() { strategy.greet(); }
};

int main() {
    SayHi sayhi;
    Ignore ignore;
    Admission admission;
    Context c1(sayhi), c2(ignore), c3(admission);
    c1.greet();
    c2.greet();
    c3.greet();
} ///:~

```

**Context::greet()** 可以正规地写得更复杂些；它类似模板方法模式，因为其中包含了不能改变的代码。但在函数 **main()** 中可以看到，可以在运行时就策略进行选择。更进一步的做法，可以将状态模式与在 **Context** 对象的生存期期间变化的策略模式结合起来使用。

## 10.10 职责链模式：尝试采用一系列策略模式

职责链（Chain of Responsibility）模式也许被看做一个使用策略对象的“递归的动态一般化”。此时提出一个调用，在一个链序列中的每个策略都试图满足这个调用。这个过程直到有一个策略成功满足该调用或者到达链序列的末尾才结束。在递归方法中，有个函数反复调用其自身直至达到某个终止条件；在职责链中，一个函数调用自身，（通过遍历策略链）调用函数的一个不同实现，如此反复直至达到某个终止条件。这个终止条件或者是已到达策略链的底部（这样就会返回一个默认对象；不管能否提供这个默认结果，必须有个方法能够决定该职责链搜索是成功还是失败）或者是成功找到一个策略。

除了调用一个函数来满足某个请求以外，链中的多个函数都有此机会满足某个请求，因此它有点专家系统的意味。由于职责链实际上就是一个链表，它能够动态创建，因此它可以看做



是一个更一般的动态构建的**switch**语句。

在GoF中，有很多关于如何将职责链模式创建一个链表的讨论。然而，如果审视这类模式，实在没必要考虑链是如何创建的；这是一个实现的细节。由于GoF是在大多数C++编译器中STL容器可利用之前编写的，它讨论创建链表最可能的原因有：（1）编译器中没有内置的链表，因此必须自己创建；（2）数据结构常常是作为学术界的一门基本的技术进行教授，GoF的作者们还没有数据结构应该是编程语言提供的有效标准工具的概念。讨论如何使用容器来实现职责链作为链的细节（在GoF中，它就是一个链表）对于这里的问题的解决没有什么意义，可以很方便地用STL容器来实现，如下所示。

在这里可以看到，使用一种自动递归搜索链中每个策略的机制，职责链模式自动找到一个解决方法：

```
//: C10:ChainOfResponsibility.cpp
// The approach of the five-year-old.
#include <iostream>
#include <vector>
#include "../purge.h"
using namespace std;

enum Answer { NO, YES };

class GimmeStrategy {
public:
    virtual Answer canIHave() = 0;
    virtual ~GimmeStrategy() {}
};

class AskMom : public GimmeStrategy {
public:
    Answer canIHave() {
        cout << "Moom? Can I have this?" << endl;
        return NO;
    }
};

class AskDad : public GimmeStrategy {
public:
    Answer canIHave() {
        cout << "Dad, I really need this!" << endl;
        return NO;
    }
};

class AskGrandpa : public GimmeStrategy {
public:
    Answer canIHave() {
        cout << "Grandpa, is it my birthday yet?" << endl;
        return NO;
    }
};

class AskGrandma : public GimmeStrategy {
public:
    Answer canIHave() {
        cout << "Grandma, I really love you!" << endl;
        return YES;
    }
};
```



```

class Gimme : public GimmeStrategy {
    vector<GimmeStrategy*> chain;
public:
    Gimme() {
        chain.push_back(new AskMom());
        chain.push_back(new AskDad());
        chain.push_back(new AskGrandpa());
        chain.push_back(new AskGrandma());
    }
    Answer canIHave() {
        vector<GimmeStrategy*>::iterator it = chain.begin();
        while(it != chain.end())
            if((*it++)->canIHave() == YES)
                return YES;
        // Reached end without success...
        cout << "Whiiiiinnne!" << endl;
        return NO;
    }
    ~Gimme() { purge(chain); }
};

int main() {
    Gimme chain;
    chain.canIHave();
} ///:~

```

注意，“语境”类**Gimme**和所有策略类都派生自同一个基类**GimmeStrategy**。

如果读者研读GoF中关于职责链的那部分内容，将会发现其结构与上面介绍的内容有明显不一致地方，因为他们专注于创建自己的链表。然而，如果牢记职责链的本质是尝试多个解决方法直到找到一个起作用的方法，读者就会了解按顺序排好的实现机制并不是该模式的本质所在。

### 10.11 工厂模式：封装对象的创建

当发现需要添加新的类型到一个系统中时，最明智的首要步骤就是用多态机制为这些新类型创建一个共同的接口。用这种方法可以将系统中其余的代码与新添加的特定类型的代码分开。新类型的添加并不会扰乱已存在的代码……或者至少看上去如此。起初它似乎只需要在继承新类的地方修改代码，但这并非完全正确。仍须创建新类型的对象，在创建对象的地方必须指定要使用的准确的构造函数。因此，如果创建对象的代码遍布整个应用程序，在增加新类型时将会遇到同样的问题——仍然必须找出代码中所有与新类型相关的地方。这是由类的创建而不是类的使用（类型的使用问题已被多态机制解决了）而引起，但是效果是一样的：添加新类型将导致问题的出现。

这个问题的解决方法就是强制用一个通用的工厂（factory）来创建对象，而不允许将创建对象的代码散布于整个系统。如果程序中所有需要创建对象的代码都转到这个工厂执行，那么在增加新对象时所要做的全部工作就是只需修改工厂。这种设计是众所周知的工厂方法（Factory Method）模式的一种变体。由于每个面向对象应用程序都需要创建对象，并且由于人们可能通过添加新类型来扩展应用程序，工厂模式可能是所有设计模式中最有用的模式之一。

举一个例子，考虑常用的**Shape**例子。实现工厂模式的一种方法就是在基类中定义一个静态成员函数：

```

//: C10:ShapeFactory1.cpp
#include <iostream>
#include <stdexcept>
#include <cstdint>

```

```

#include <string>
#include <vector>
#include "../purge.h"
using namespace std;

class Shape {
public:
    virtual void draw() = 0;
    virtual void erase() = 0;
    virtual ~Shape() {}
    class BadShapeCreation : public logic_error {
    public:
        BadShapeCreation(string type)
            : logic_error("Cannot create type " + type) {}
    };
    static Shape* factory(const string& type)
        throw(BadShapeCreation);
};

class Circle : public Shape {
    Circle() {} // Private constructor
    friend class Shape;
public:
    void draw() { cout << "Circle::draw" << endl; }
    void erase() { cout << "Circle::erase" << endl; }
    ~Circle() { cout << "Circle::~~Circle" << endl; }
};

class Square : public Shape {
    Square() {}
    friend class Shape;
public:
    void draw() { cout << "Square::draw" << endl; }
    void erase() { cout << "Square::erase" << endl; }
    ~Square() { cout << "Square::~~Square" << endl; }
};

Shape* Shape::factory(const string& type)
    throw(Shape::BadShapeCreation) {
    if(type == "Circle") return new Circle;
    if(type == "Square") return new Square;
    throw BadShapeCreation(type);
}

char* sl[] = { "Circle", "Square", "Square",
               "Circle", "Circle", "Circle", "Square" };
int main() {
    vector<Shape*> shapes;
    try {
        for(size_t i = 0; i < sizeof sl / sizeof sl[0]; i++)
            shapes.push_back(Shape::factory(sl[i]));
    } catch(Shape::BadShapeCreation e) {
        cout << e.what() << endl;
        purge(shapes);
        return EXIT_FAILURE;
    }
    for(size_t i = 0; i < shapes.size(); i++) {
        shapes[i]->draw();
        shapes[i]->erase();
    }
    purge(shapes);
} ///:~

```



函数 **factory()** 允许以一个参数来决定创建何种类型的 **Shape**。在这里，参数类型为 **string**，也可以是任何数据集。在添加新的 **Shape** 类型时，函数 **factory()** 是当前系统中惟一需要修改的代码。（对象的初始化数据大概也可以由系统外获得，而不必像本例中那样来自硬编码数组。）

为了确保对象的创建只能发生在函数 **factory()** 中，**Shape** 的特定类型的构造函数被设为私有，同时 **Shape** 被声明为友元类，因此 **factory()** 能够访问这些构造函数。（也可以只将 **Shape::factory()** 声明为友元函数，但是似乎声明整个基类为友元类也没什么大碍。）这样的设计还有另外一个重要的含义——基类 **Shape** 现在必须了解每个派生类的细节——这是面向对象设计试图避免的一个性质。对于结构框架或者任何类库来说都应该支持扩充，但这样一来，系统很快就会变得笨拙，因为一旦新类型被加到这种层次结构中，基类就必须更新。可以使用下一小节将要讨论的多态工厂（polymorphic factory）来避免这种循环依赖。

### 10.11.1 多态工厂

在前面的例子中，静态成员函数 **static factory()** 迫使所有创建对象的操作都集中在一个地方，因此这个地方就是惟一需要修改代码的地方。这确实是一个合理的解决方法，因为它完美地封装了对象的创建过程。然而，“四人帮”强调工厂方法模式的理由是，可以使不同类型的工厂派生自基本类型的工厂。工厂方法模式事实上是多态工厂模式的一个特例。这里修改了 **ShapeFactory1.cpp**，所以工厂方法模式作为一个单独的类中的虚函数出现：

```
//: C10:ShapeFactory2.cpp
// Polymorphic Factory Methods.
#include <iostream>
#include <map>
#include <string>
#include <vector>
#include <stdexcept>
#include <cstddef>
#include "../purge.h"
using namespace std;

class Shape {
public:
    virtual void draw() = 0;
    virtual void erase() = 0;
    virtual ~Shape() {}
};

class ShapeFactory {
    virtual Shape* create() = 0;
    static map<string, ShapeFactory*> factories;
public:
    virtual ~ShapeFactory() {}
    friend class ShapeFactoryInitializer;
    class BadShapeCreation : public logic_error {
    public:
        BadShapeCreation(string type)
            : logic_error("Cannot create type " + type) {}
    };
    static Shape*
    createShape(const string& id) throw(BadShapeCreation) {
        if(factories.find(id) != factories.end())
            return factories[id]->create();
        else
            throw BadShapeCreation(id);
    }
};
```



```

    }
};
// Define the static object:
map<string, ShapeFactory*> ShapeFactory::factories;

class Circle : public Shape {
    Circle() {} // Private constructor
    friend class ShapeFactoryInitializer;
    class Factory;
    friend class Factory;
    class Factory : public ShapeFactory {
    public:
        Shape* create() { return new Circle; }
        friend class ShapeFactoryInitializer;
    };
public:
    void draw() { cout << "Circle::draw" << endl; }
    void erase() { cout << "Circle::erase" << endl; }
    ~Circle() { cout << "Circle::~~Circle" << endl; }
};

class Square : public Shape {
    Square() {}
    friend class ShapeFactoryInitializer;
    class Factory;
    friend class Factory;
    class Factory : public ShapeFactory {
    public:
        Shape* create() { return new Square; }
        friend class ShapeFactoryInitializer;
    };
public:
    void draw() { cout << "Square::draw" << endl; }
    void erase() { cout << "Square::erase" << endl; }
    ~Square() { cout << "Square::~~Square" << endl; }
};

// Singleton to initialize the ShapeFactory:
class ShapeFactoryInitializer {
    static ShapeFactoryInitializer si;
    ShapeFactoryInitializer() {
        ShapeFactory::factories["Circle"] = new Circle::Factory;
        ShapeFactory::factories["Square"] = new Square::Factory;
    }
    ~ShapeFactoryInitializer() {
        map<string, ShapeFactory*>::iterator it =
            ShapeFactory::factories.begin();
        while(it != ShapeFactory::factories.end())
            delete it++->second;
    }
};

// Static member definition:
ShapeFactoryInitializer ShapeFactoryInitializer::si;

char* sl[] = { "Circle", "Square", "Square",
    "Circle", "Circle", "Circle", "Square" };

int main() {
    vector<Shape*> shapes;
    try {
        for(size_t i = 0; i < sizeof sl / sizeof sl[0]; i++)

```



```

        shapes.push_back(ShapeFactory::createShape(sl[i]));
    } catch(ShapeFactory::BadShapeCreation e) {
        cout << e.what() << endl;
        return EXIT_FAILURE;
    }
    for(size_t i = 0; i < shapes.size(); i++) {
        shapes[i]->draw();
        shapes[i]->erase();
    }
    purge(shapes);
} ///:~

```

现在，工厂方法模式作为**virtual create()**出现在它自己的**ShapeFactory**类中。这是一个私有成员函数，意味着不能直接调用它，但可以被覆盖。**Shape**的子类必须创建各自的**ShapeFactory**子类，并且覆盖成员函数**create()**以创建其自身类型的对象。这些工厂是私有的，只能被主工厂方法模式访问。采用这种方法，所有客户代码都必须通过工厂方法模式创建对象。

**Shape**对象的实际创建是通过调用**ShapeFactory::createShape()**完成的，这是一个静态成员函数，使用**ShapeFactory**中的**map**根据传递给它的标识符找到相应的工厂对象。工厂直接创建**Shape**对象，但是可以设想一个更为复杂的问题：在某个地方返回一个合适的工厂对象，然后该工厂对象被调用者用于以更复杂的方法创建一个对象。然而，似乎在大多数情况下不需要这么复杂地使用多态工厂方法模式，基类中的一个静态成员函数（正如**ShapeFactory1.cpp**中所示）就能很好地完成这项工作。

注意，**ShapeFactory**必须通过装载它的**map**与工厂对象进行初始化，这些操作发生在单件**ShapeFactoryInitializer**中。当增加一个新类型到这个设计时，必须定义该类型，创建一个工厂并修改**ShapeFactoryInitializer**，以便将工厂的一个实例插入**map**中。这些额外的复杂操作再次暗示，如果不需要创建独立的工厂对象，尽可能使用静态（**static**）工厂方法模式。

### 10.11.2 抽象工厂

抽象工厂（Abstract Factory）模式看起来和前面看到的工厂方法很相似，只是它使用若干工厂方法（Factory Method）模式。每个工厂方法模式创建一个不同类型的对象。当创建一个工厂对象时，要决定将如何使用由那个工厂创建的所有对象。“四人帮”书中的例子实现各种图形用户界面（GUI）的可移植性：创建一个适合于正在使用的GUI的工厂对象，然后它将根据对它发出的对一个菜单、按钮或者滚动条等的请求自动创建适合该GUI的项目版本。这样就能够在一个地方隔离从一个GUI转变到另一个GUI的作用。

再举一个例子，假设要创建一个通用的游戏环境，并且希望它能支持不同类型的游戏。请看以下程序是如何使用抽象工厂模式的：

```

//: C10:AbstractFactory.cpp
// A gaming environment.
#include <iostream>
using namespace std;

class Obstacle {
public:
    virtual void action() = 0;
};

class Player {
public:
    virtual void interactWith(Obstacle*) = 0;

```

```

};

class Kitty: public Player {
    virtual void interactWith(Obstacle* ob) {
        cout << "Kitty has encountered a ";
        ob->action();
    }
};

class KungFuGuy: public Player {
    virtual void interactWith(Obstacle* ob) {
        cout << "KungFuGuy now battles against a ";
        ob->action();
    }
};

class Puzzle: public Obstacle {
public:
    void action() { cout << "Puzzle" << endl; }
};

class NastyWeapon: public Obstacle {
public:
    void action() { cout << "NastyWeapon" << endl; }
};

// The abstract factory:
class GameElementFactory {
public:
    virtual Player* makePlayer() = 0;
    virtual Obstacle* makeObstacle() = 0;
};

// Concrete factories:
class KittiesAndPuzzles : public GameElementFactory {
public:
    virtual Player* makePlayer() { return new Kitty; }
    virtual Obstacle* makeObstacle() { return new Puzzle; }
};

class KillAndDismember : public GameElementFactory {
public:
    virtual Player* makePlayer() { return new KungFuGuy; }
    virtual Obstacle* makeObstacle() {
        return new NastyWeapon;
    }
};

class GameEnvironment {
    GameElementFactory* gef;
    Player* p;
    Obstacle* ob;
public:
    GameEnvironment(GameElementFactory* factory)
        : gef(factory), p(factory->makePlayer()),
          ob(factory->makeObstacle()) {}
    void play() { p->interactWith(ob); }
    ~GameEnvironment() {
        delete p;
        delete ob;
        delete gef;
    }
};

```



```

int main() {
    GameEnvironment
        gl(new KittiesAndPuzzles),
        g2(new KillAndDismember);
    gl.play();
    g2.play();
}
/* Output:
Kitty has encountered a Puzzle
KungFuGuy now battles against a NastyWeapon */ ///:~

```

在此环境中，**Player**对象与**Obstacle**对象交互，但是**Player**和**Obstacle**类型依赖于具体的游戏。可以选择特定的**GameElementFactory**来决定游戏的类型，然后**GameEnvironment**控制游戏的设置和进行。在本例中，游戏的设置和进行很简单，但是那些动作（初始条件（initial condition）和状态变化（state change））在很大程度上决定了游戏的结果。在这里，**GameEnvironment**不是设计成继承的，即使这样做可能是有意义的。

这个例子也说明将在稍后讨论双重派遣（double dispatching）。

### 10.11.3 虚构造函数

使用工厂方法模式的主要目标之一就是更好地组织代码，使得在创建对象时不需要选择准确的构造函数类型。也就是说，可以告诉工厂：“现在还不能确切地知道需要什么类型的对象，但是这里有一些信息。请创建类型适当的对象。”

此外，在构造函数调用期间，虚拟机制并不起作用（发生早期绑定）。在某些情况下这是很棘手的事情。例如，在**Shape**程序中，在**Shape**对象的构造函数内部建立一切需要的东西然后由**draw()**绘制**Shape**，这似乎是合理的。函数**draw()**应该是一个虚函数，它将根据传递给**Shape**的消息绘制相应的图形，消息表明图形本身是**Circle**、**Square**或者**Line**。然而，这些操作在构造函数内部不能采用这种方法，因为当在构造函数内部调用虚函数时，将由虚函数决定指向哪个“局部的”函数体。

如果想要在构造函数中调用虚函数，并使其完成正确的工作，必须使用某种技术来模拟虚构造函数。这是一个难题。请记住，虚函数的思想就是发送一个消息给对象，而让对象确定要做的正确事情。但是对象是由构造函数创建的。因此，虚构造函数好像是在对一个对象说：“我不能准确知道你是什么类型的对象，但是无论如何要以正确的类型建造你。”对于普通的构造函数来说，编译器在编译时必须知道虚指针（VPTR）指向的虚函数表（VTABLE）的地址；而对于虚构造函数，即使存在这样的虚函数表，它也不可能做到这一点，因为它在编译时不知道任何类型信息。构造函数不能为虚函数是有道理的，因为它是这样一种函数，必须完全知道有关对象类型的所有信息。

可是，程序员有时还想要得到接近于虚构造函数的行为。

在**Shape**的例子中，在参数表中对**Shape**构造函数提交一些特定的信息，使构造函数创建特定类型的**Shape**对象（一个**Circle**或是一个**Square**）而无须更多的干涉，这将是很好的。通常，程序员自己必需显式调用**Circle**或是**Square**的构造函数。

Coplien<sup>①</sup>将他给出的解决此问题的方法取名为“信封和信件类”。“信封”类是基类，它是一个包含指向一个对象的指针的外壳，该对象也是一个基类类型。“信封”类的构造函数决定采用什么样的特定类型，在堆上创建一个该类型的对象，然后对它的指针分配对象（决定是在

① James O. Coplien, 《Advanced C++ Programming Styles and Idioms》, Addison Wesley, 1992.



运行中调用构造函数时做出的，而不是在编译中做类型正常检查时做出的)。随后的所有函数调用都是由基类通过它的指针来进行处理。这实际上就是状态模式的小小变形，其中基类扮演派生类的代理的角色，而派生类提供行为中的变化：

```

//: C10:VirtualConstructor.cpp
#include <iostream>
#include <string>
#include <stdexcept>
#include <stdexcept>
#include <cstdint>
#include <vector>
#include "../purge.h"
using namespace std;

class Shape {
    Shape* s;
    // Prevent copy-construction & operator=
    Shape(Shape&);
    Shape operator=(Shape&);
protected:
    Shape() { s = 0; }
public:
    virtual void draw() { s->draw(); }
    virtual void erase() { s->erase(); }
    virtual void test() { s->test(); }
    virtual ~Shape() {
        cout << "~Shape" << endl;
        if(s) {
            cout << "Making virtual call: ";
            s->erase(); // Virtual call
        }
        cout << "delete s: ";
        delete s; // The polymorphic deletion
        // (delete 0 is legal; it produces a no-op)
    }
    class BadShapeCreation : public logic_error {
    public:
        BadShapeCreation(string type)
            : logic_error("Cannot create type " + type) {}
    };
    Shape(string type) throw(BadShapeCreation);
};

class Circle : public Shape {
    Circle(Circle&);
    Circle operator=(Circle&);
    Circle() {} // Private constructor
    friend class Shape;
public:
    void draw() { cout << "Circle::draw" << endl; }
    void erase() { cout << "Circle::erase" << endl; }
    void test() { draw(); }
    ~Circle() { cout << "Circle::~~Circle" << endl; }
};

class Square : public Shape {
    Square(Square&);
    Square operator=(Square&);
    Square() {}
    friend class Shape;
public:

```



```

void draw() { cout << "Square::draw" << endl; }
void erase() { cout << "Square::erase" << endl; }
void test() { draw(); }
~Square() { cout << "Square::~Square" << endl; }
};

Shape::Shape(string type) throw(Shape::BadShapeCreation) {
    if(type == "Circle")
        s = new Circle;
    else if(type == "Square")
        s = new Square;
    else throw BadShapeCreation(type);
    draw(); // Virtual call in the constructor
}

char* sl[] = { "Circle", "Square", "Square",
               "Circle", "Circle", "Circle", "Square" };

int main() {
    vector<Shape*> shapes;
    cout << "virtual constructor calls:" << endl;
    try {
        for(size_t i = 0; i < sizeof sl / sizeof sl[0]; i++)
            shapes.push_back(new Shape(sl[i]));
    } catch(Shape::BadShapeCreation e) {
        cout << e.what() << endl;
        purge(shapes);
        return EXIT_FAILURE;
    }
    for(size_t i = 0; i < shapes.size(); i++) {
        shapes[i]->draw();
        cout << "test" << endl;
        shapes[i]->test();
        cout << "end test" << endl;
        shapes[i]->erase();
    }
    Shape c("Circle"); // Create on the stack
    cout << "destructor calls:" << endl;
    purge(shapes);
} ///:~

```

基类**Shape**包含一个对象指针作为其惟一的数据成员，该指针指向**Shape**类型的对象。（在创建一个“虚构造函数”的模式时，务必确保这个指针总是被初始化成指向一个激活的对象。）这个基类实际上就是一个代理，因为这是客户程序惟一所能看到和与之进行交互的对象。

每次从**Shape**派生新的子类时，必须回到基类并且在基类**Shape**的“虚构造函数”内的一个位置增加那个类型的创建。这并不是件很繁重的任务，但缺点是在**Shape**类和其所有的派生类之间形成了依赖关系。

在这个例子中，交给虚构造函数的关于要创建对象的类型信息必须是显式说明的：它是一个用来命名类型的**string**。但是模式也可以用其他信息——比如说，在一个语法分析器中，可以把扫描器的输出结果给虚构造函数，而构造函数将利用这些信息来决定创建何种类型的对象。

虚构造函数**Shape(type)**在所有派生类未声明前不能定义。然而，默认的构造函数能够在**class Shape**中定义，但是它应该被声明为**protected**的，所以不能创建临时的**Shape**对象。这个默认的构造函数只能被派生类对象的构造函数调用。程序员被迫显式地创建一个默认构造函数，因为如果没有定义构造函数编译器将会自动创建一个。因为必须定义**Shape(type)**，所以也必须定义**Shape()**。

在这种模式中，默认构造函数至少有一个重要的工作要做——它必须将指针**s**设置为零值。这在初听起来有点奇怪，但是应当记得，默认构造函数将作为实际对象的构造的一部分被调用——用Coplien的术语来说，它是“信件”而不是“信封”。然而，“信件”也是从“信封”派生出来的，它也继承数据成员**s**。在“信封”中**s**很重要，因为它指向实际的对象，但是在“信件”中，**s**只是一个超重行李。可是，即便是额外行李也应该被初始化。如果不调用默认构造函数为“信件”把**s**赋为零值，将会出现问题（这在后面将会看到）。

虚构造函数使用其参数提供的信息，这些信息完全能够决定对象的类型。注意，这些类型信息在运行时才能读取和使用，而在一般情况下，编译器在编译时必须知道确切的类型（这是本系统能够有效地模拟虚构造函数的另外一个原因）。

虚构造函数使用其参数来选择要构造的实际对象（“信件”），然后对“信封”内的指针赋值。至此，“信件”类对象创建完成，因此任何虚函数的调用得以正确地重定向。

作为一个例子，考虑在虚构造函数中调用函数**draw()**。如果跟踪这个调用（手工或者使用调试器），将会看到它是从基类**Shape**中的函数**draw()**开始调用的。这个函数调用“信封”的**draw()**，“信封”指针**s**指向它的“信件”。所有从**Shape**中派生出来的类型共享同一个接口，所以这个虚调用能够正确执行，虽然它似乎在构造函数中。（实际上，“信件”类的构造函数已经执行完毕。）只要基类中所有的虚调用通过这个指向“信件”的指针仅调用同一个虚函数，系统就能正确地运作。

为了了解它是如何工作的，请思考**main()**函数中的代码。为了填充**vector shapes**，调用“虚构造函数”以便产生**Shape**对象。通常像这样的情况，应该用调用实际类型的构造函数，这种类型的虚指针（VPTR）应该安置在该对象中。然而在这里，在每种情况下虚指针（VPTR）都是指向**Shape**的一个对象，而不是指向特定的一种类型如**Circle**、**Square**或是**Triangle**。

在**for**循环中，为每个**Shape**对象调用函数**draw()**和**erase()**，虚函数调用通过VPTR解析到相应类型。然而，在各种情况下它都是**Shape**。事实上，读者也许想知道为什么**draw()**和**erase()**要声明为虚函数。在下一步可以看到原因：**draw()**的基类版本通过“信件”指针**s**调用“信件”的虚函数**draw()**。这时，这个调用解析到对象的实际类型，而不是基类**Shape**。因此每次调用虚函数时，使用虚构造函数的运行时代价只是一个额外的虚间接引用（virtual indirection）。

为了创建如**draw()**、**erase()**和**test()**等任何将被覆盖的函数，如前所述，必须全部前向调用基类实现中的指针**s**。这是因为当调用发生时，调用“信封”的成员函数将被解析指向**Shape**而不是**Shape**的派生类。只有当前向调用的时候，**s**才发生虚行为。在**main()**函数中，可以看到所有工作都能正确执行，即使调用发生在构造函数和析构函数中。

#### 析构函数操作

在这种模式中析构活动同样也是很复杂的。为了了解这点，让我们从头至尾说明，当对指向创建在堆上的一个**Shape**对象（尤其是一个**Square**）的指针调用**delete**时会发生什么情况。（这是比建立在栈上的对象复杂得多的对象。）这将是经过多态接口的**delete**，并通过调用**purge()**来完成。

**Shapes**中任何指针的类型都是基类**Shape**的类型，所以编译器通过**Shape**产生调用。通常情况下，可以说这是一个虚调用，所以**Square**的析构函数将被调用。但是在用虚构造函数系统中，由编译器来创建实际的**Shape**对象，即使构造函数初始化“信件”的指针为指向一个特定的**Shape**类型。这里使用了虚机制，但是，在**Shape**对象中的VPTR是**Shape**对象的虚指针，而不是**Square**对象的虚指针。这样就解析到**Shape**的析构函数，该析构函数调用

**delete**，该指针实际指向一个**Square**对象。这还是个虚调用，不过这时它解析指向**Square**对象的析构函数。

C++通过编译器确保继承层次结构中的所有析构函数都被调用。**Square**的析构函数最先被调用，然后顺序调用任何中间类的析构函数，直至最后，基类的析构函数被调用。这个基类的析构函数中包含代码**delete s**。当这个析构函数最初被调用时，它针对的是“信封”的**s**，而现在它针对的是“信件”的**s**，这是因为“信件”从“信封”中继承，而不是因为它包含了什么东西。所以这个**delete**调用不应该有任何操作。

解决此问题的方法是使“信件”的指针**s**指向零。这样，当调用“信件”的基类析构函数时，实际上得到的就是**delete o**，它的定义是不执行任何操作。因为默认构造函数被设为保护的，它只是在“信件”对象的构造过程中被调用。这是将**s**置为零值的惟一情况。

虽然这种描述很有趣，但是可以看到这是一个复杂的方法，所以隐藏构造的最常见的工具一般是普通的“工厂方法”而不是“虚构造函数”模式这样的方法。

## 10.12 构建器模式：创建复杂对象

构建器（Builder）（它和前面已经讨论过的工厂方法一样，属于创建型模式）模式的目标是将对象的创建与它的“表示法”（representation）分开。这就意味着，创建过程保持原状，但是产生对象的表示法可能不同。“四人帮”指出，构建器模式和抽象工厂模式主要的区别就是，构建器模式一步步创建对象，所以及时展开输出创建过程就似乎很重要。此外，“主管（director）”获得一个切片的流（stream），并且将这些切片传递给构建器，每个切片用来执行创建过程中的一步。

下面有一个例子，作为模型的一辆自行车按照其类型（山地车、旅行车或赛车）来选择零部件组装一辆自行车。一个构建器与每个自行车类都关联，每个构建器实现的接口由抽象类**BicycleBuilder**中指定。单独的类**BicycleTechnician**表示“四人帮”中描述的“导向器”对象，它使用具体的**BicycleBuilder**对象来构造**Bicycle**对象。

```
//: C10:Bicycle.h
// Defines classes to build bicycles;
// Illustrates the Builder design pattern.
#ifndef BICYCLE_H
#define BICYCLE_H
#include <iostream>
#include <string>
#include <vector>
#include <cstdint>
#include "../purge.h"
using std::size_t;

class BicyclePart {
public:
    enum BPart { FRAME, WHEEL, SEAT, DERAILLEUR,
                HANDLEBAR, SPROCKET, RACK, SHOCK, NPARTS };
private:
    BPart id;
    static std::string names[NPARTS];
public:
    BicyclePart(BPart bp) { id = bp; }
    friend std::ostream&
    operator<<(std::ostream& os, const BicyclePart& bp) {
        return os << bp.names[bp.id];
    }
}
```



```

};

class Bicycle {
    std::vector<BicyclePart*> parts;
public:
    ~Bicycle() { purge(parts); }
    void addPart(BicyclePart* bp) { parts.push_back(bp); }
    friend std::ostream&
    operator<<(std::ostream& os, const Bicycle& b) {
        os << "{ ";
        for(size_t i = 0; i < b.parts.size(); ++i)
            os << *b.parts[i] << ' ';
        return os << '}';
    }
};

class BicycleBuilder {
protected:
    Bicycle* product;
public:
    BicycleBuilder() { product = 0; }
    void createProduct() { product = new Bicycle; }
    virtual void buildFrame() = 0;
    virtual void buildWheel() = 0;
    virtual void buildSeat() = 0;
    virtual void buildDeraillleur() = 0;
    virtual void buildHandlebar() = 0;
    virtual void buildSprocket() = 0;
    virtual void buildRack() = 0;
    virtual void buildShock() = 0;
    virtual std::string getBikeName() const = 0;
    Bicycle* getProduct() {
        Bicycle* temp = product;
        product = 0; // Relinquish product
        return temp;
    }
};

class MountainBikeBuilder : public BicycleBuilder {
public:
    void buildFrame();
    void buildWheel();
    void buildSeat();
    void buildDeraillleur();
    void buildHandlebar();
    void buildSprocket();
    void buildRack();
    void buildShock();
    std::string getBikeName() const { return "MountainBike"; }
};

class TouringBikeBuilder : public BicycleBuilder {
public:
    void buildFrame();
    void buildWheel();
    void buildSeat();
    void buildDeraillleur();
    void buildHandlebar();
    void buildSprocket();
    void buildRack();
    void buildShock();
    std::string getBikeName() const { return "TouringBike"; }
};

```



```

class RacingBikeBuilder : public BicycleBuilder {
public:
    void buildFrame();
    void buildWheel();
    void buildSeat();
    void buildDeraillieur();
    void buildHandlebar();
    void buildSprocket();
    void buildRack();
    void buildShock();
    std::string getBikeName() const { return "RacingBike"; }
};

class BicycleTechnician {
    BicycleBuilder* builder;
public:
    BicycleTechnician() { builder = 0; }
    void setBuilder(BicycleBuilder* b) { builder = b; }
    void construct();
};
#endif // BICYCLE_H ///:~

```

**Bicycle**持有一个**vector**，用于保存指向**BicyclePart**对象的指针，这些对象表示用于构造自行车的部件。由一个**BicycleTechnician**（本例中的“主管”）调用派生的**BicycleBuilder**对象的函数**BicycleBuilder::createproduct()**来初始化一辆自行车的创建。**BicycleTechnician::construct()**函数调用**BicycleBuilder**接口中的所有函数（因为它不知道有什么具体的构建器类型）。具体的构建器类省略了（通过空函数体）那些与他们所构建的自行车的类型无关的动作，如下面的实现文件所示：

```

//: C10:Bicycle.cpp {0} {-mwcc}
#include "Bicycle.h"
#include <cassert>
#include <cstdint>
using namespace std;

std::string BicyclePart::names[NPARTS] = {
    "Frame", "Wheel", "Seat", "Deraillieur",
    "Handlebar", "Sprocket", "Rack", "Shock" };

// MountainBikeBuilder implementation
void MountainBikeBuilder::buildFrame() {
    product->addPart(new BicyclePart(BicyclePart::FRAME));
}
void MountainBikeBuilder::buildWheel() {
    product->addPart(new BicyclePart(BicyclePart::WHEEL));
}
void MountainBikeBuilder::buildSeat() {
    product->addPart(new BicyclePart(BicyclePart::SEAT));
}
void MountainBikeBuilder::buildDeraillieur() {
    product->addPart(
        new BicyclePart(BicyclePart::DERAILLEUR));
}
void MountainBikeBuilder::buildHandlebar() {
    product->addPart(
        new BicyclePart(BicyclePart::HANDLEBAR));
}
void MountainBikeBuilder::buildSprocket() {
    product->addPart(new BicyclePart(BicyclePart::SPROCKET));
}

```

```

void MountainBikeBuilder::buildRack() {}
void MountainBikeBuilder::buildShock() {
    product->addPart(new BicyclePart(BicyclePart::SHOCK));
}

// TouringBikeBuilder implementation
void TouringBikeBuilder::buildFrame() {
    product->addPart(new BicyclePart(BicyclePart::FRAME));
}
void TouringBikeBuilder::buildWheel() {
    product->addPart(new BicyclePart(BicyclePart::WHEEL));
}
void TouringBikeBuilder::buildSeat() {
    product->addPart(new BicyclePart(BicyclePart::SEAT));
}
void TouringBikeBuilder::buildDeraillieur() {
    product->addPart(
        new BicyclePart(BicyclePart::DERAILLEUR));
}
void TouringBikeBuilder::buildHandlebar() {
    product->addPart(
        new BicyclePart(BicyclePart::HANDLEBAR));
}
void TouringBikeBuilder::buildSprocket() {
    product->addPart(new BicyclePart(BicyclePart::SPROCKET));
}
void TouringBikeBuilder::buildRack() {
    product->addPart(new BicyclePart(BicyclePart::RACK));
}
void TouringBikeBuilder::buildShock() {}

// RacingBikeBuilder implementation
void RacingBikeBuilder::buildFrame() {
    product->addPart(new BicyclePart(BicyclePart::FRAME));
}
void RacingBikeBuilder::buildWheel() {
    product->addPart(new BicyclePart(BicyclePart::WHEEL));
}
void RacingBikeBuilder::buildSeat() {
    product->addPart(new BicyclePart(BicyclePart::SEAT));
}
void RacingBikeBuilder::buildDeraillieur() {}
void RacingBikeBuilder::buildHandlebar() {
    product->addPart(
        new BicyclePart(BicyclePart::HANDLEBAR));
}
void RacingBikeBuilder::buildSprocket() {
    product->addPart(new BicyclePart(BicyclePart::SPROCKET));
}
void RacingBikeBuilder::buildRack() {}
void RacingBikeBuilder::buildShock() {}

// BicycleTechnician implementation
void BicycleTechnician::construct() {
    assert(builder);
    builder->createProduct();
    builder->buildFrame();
    builder->buildWheel();
    builder->buildSeat();
    builder->buildDeraillieur();
    builder->buildHandlebar();
    builder->buildSprocket();
    builder->buildRack();
}

```



```

    builder->buildShock();
} ///:~

```

**Bicycle**流插入符为各个**BicyclePart**调用相应的插入符，并且打印其类型名称以便知道**Bicycle**对象包含的内容。程序举例如下：

```

//: C10:BuildBicycles.cpp
//{L} Bicycle
// The Builder design pattern.
#include <cstdint>
#include <iostream>
#include <map>
#include <vector>
#include "Bicycle.h"
#include "../purge.h"
using namespace std;

// Constructs a bike via a concrete builder
Bicycle* buildMeABike(
    BicycleTechnician& t, BicycleBuilder* builder) {
    t.setBuilder(builder);
    t.construct();
    Bicycle* b = builder->getProduct();
    cout << "Built a " << builder->getBikeName() << endl;
    return b;
}

int main() {
    // Create an order for some bicycles
    map<string, size_t> order;
    order["mountain"] = 2;
    order["touring"] = 1;
    order["racing"] = 3;

    // Build bikes
    vector<Bicycle*> bikes;
    BicycleBuilder* m = new MountainBikeBuilder;
    BicycleBuilder* t = new TouringBikeBuilder;
    BicycleBuilder* r = new RacingBikeBuilder;
    BicycleTechnician tech;
    map<string, size_t>::iterator it = order.begin();
    while(it != order.end()) {
        BicycleBuilder* builder;
        if(it->first == "mountain")
            builder = m;
        else if(it->first == "touring")
            builder = t;
        else if(it->first == "racing")
            builder = r;
        for(size_t i = 0; i < it->second; ++i)
            bikes.push_back(buildMeABike(tech, builder));
        ++it;
    }
    delete m;
    delete t;
    delete r;

    // Display inventory
    for(size_t i = 0; i < bikes.size(); ++i)
        cout << "Bicycle: " << *bikes[i] << endl;
    purge(bikes);
}

```





```

/* Output:
Built a MountainBike
Built a MountainBike
Built a RacingBike
Built a RacingBike
Built a RacingBike
Built a TouringBike
Bicycle: {
    Frame Wheel Seat Derailleur Handlebar Sprocket Shock }
Bicycle: {
    Frame Wheel Seat Derailleur Handlebar Sprocket Shock }
Bicycle: { Frame Wheel Seat Handlebar Sprocket }
Bicycle: { Frame Wheel Seat Handlebar Sprocket }
Bicycle: { Frame Wheel Seat Handlebar Sprocket }
Bicycle: {
    Frame Wheel Seat Derailleur Handlebar Sprocket Rack }
*/ ///:~

```

这种模式的功能就是它将部件组合成为一个完整产品的算法与部件本身分开，这样就允许通过一个共同接口的不同实现来为不同的产品提供不同的算法。

### 10.13 观察者模式

观察者 (Observer) 模式用于解决一个相当常见的问题：当某些其他对象改变状态时，如果一组对象需要进行相应的更新，那么应该如何处理呢？这可以在Smalltalk的MVC (model-view-controller, 模型-视图-控制器) 的“模型-视图”或是几乎完全等价的“文档-视图设计模式”中见到。假定有一些数据 (即“文档”) 和两个视图：一个图形视图和一个文本视图。在更改“文档”数据时，必须通知这些视图更新它们自身，这就是观察者模式所要完成的任务。

在下面的代码中使用两种对象的类型以实现观察者模式。类**Observable**跟踪那些当一类对象发生某种变化时需要被通知的对象。类**Observable**为列表上的每个观察者调用成员函数**notifyObservers()**。成员函数**notifyObservers()**是基类**Observable**的一部分。

在观察者模式中有两个“变化的事件”：正在进行观察的对象的数量和更新发生的方式。这就是说，观察者模式允许修改这二者而不影响周围的其他代码。

可以用很多方法来实现观察者模式，下面的代码将创建一个程序框架，读者可根据这个框架构建自己的观察者模式代码。首先，这个接口描述了什么是观察者模式，如下所示：

```

//: C10:Observer.h
// The Observer interface.
#ifndef OBSERVER_H
#define OBSERVER_H

class Observable;
class Argument {};

class Observer {
public:
    // Called by the observed object, whenever
    // the observed object is changed:
    virtual void update(Observable* o, Argument* arg) = 0;
    virtual ~Observer() {}
};
#endif // OBSERVER_H ///:~

```

因为在这种方法中**Observer**与**Observable**交互作用，所以必须首先声明**Observable**。另外，类**Argument**是空的，在更新过程中它只担任一个基类的角色，用于传递需要的任何

参数类型。如果需要，也可以仅传递额外的像**void\***类型这样的参数。在这两种情况下无论哪种情况都有向下类型转换的操作。

类**Observer**是只有一个成员函数**update()**的“接口”类。当正在被观察的对象认为到了更新其所有观察者的时机时，它将调用此函数。函数的参数是可选的；可以调用一个没有参数的**update()**，这仍然符合观察者模式的要求。然而，更常见的是——它允许被观察的对象传递引起更新操作的对象（因为一个**Observer**可以注册到多个被观察对象）和任何额外的有用信息，而不必强迫**Observer**对象自己去搜寻正在被更新的对象并取得任何其他所需的信息。

“被观察对象”的类型是**Observable**的类型：

```

//: C10:Observable.h
// The Observable class.
#ifndef OBSERVABLE_H
#define OBSERVABLE_H
#include <set>
#include "Observer.h"

class Observable {
    bool changed;
    std::set<Observer*> observers;
protected:
    virtual void setChanged() { changed = true; }
    virtual void clearChanged() { changed = false; }
public:
    virtual void addObserver(Observer& o) {
        observers.insert(&o);
    }
    virtual void deleteObserver(Observer& o) {
        observers.erase(&o);
    }
    virtual void deleteObservers() {
        observers.clear();
    }
    virtual int countObservers() {
        return observers.size();
    }
    virtual bool hasChanged() { return changed; }
    // If this object has changed, notify all
    // of its observers:
    virtual void notifyObservers(Argument* arg = 0) {
        if(!hasChanged()) return;
        clearChanged(); // Not "changed" anymore
        std::set<Observer*>::iterator it;
        for(it = observers.begin(); it != observers.end(); it++)
            (*it)->update(this, arg);
    }
    virtual ~Observable() {}
};
#endif // OBSERVABLE_H ///:~

```

再次说明，这里的设计比实际必需的更精细些。只要有方法用**Observable**注册一个**Observer**并有方法为**Observable**更新其**Observer**，不必太在意其成员函数的设定。然而，这个设计的意图是可重用的。（它是从用于Java标准库的设计中摘取出来的。）<sup>⊖</sup>

**Observable**对象有一个用于指示是否已被修改的标志。在一个简单的设计中可能没有

⊖ 它与Java不同，因为在通知所有观察者之前**java.util.Observable.notifyObservers()**不会调用**clearChanged()**。

标志；在出现变化时所有对象都将得到通知。然而需要注意的是，标志状态的控制是 **protected**，所以只有继承者才能决定是什么造成了这个变化，而不是产生派生 **Observer** 类的末端用户。

收集到的 **Observer** 对象被保存在一个 **set<Observer\*>** 中以防止复制；集合中的 **setinsert()**、**erase()**、**clear()** 和 **size()** 函数是开放的，允许在任何时候添加和删除 **Observer** 对象，因此提供了运行时的灵活性。

大部分工作是在函数 **notifyObservers()** 中做的。如果标志 **changed** 没有设置，它不做任何动作。否则，它将首先清除标志 **changed** 以防重复调用 **notifyObservers()** 所造成的时间浪费。这些是在通知观察者之前做的，以防被调用的 **update()** 会执行某些能够引起变化的操作，进而把这个变化反馈给 **Observable** 对象。然后它遍历集合 **set** 并回调每个 **Observer** 对象的成员函数 **update()**。

起初似乎可以使用一个普通的 **Observable** 对象来管理更新操作，但这不会起作用；为了使之生效，必须从 **Observable** 派生出子类并且在派生类的代码中某处调用函数 **setChanged()**。这就是设置标志“changed”的成员函数，这就意味着当调用 **notifyObservers()** 时，事实上所有观测者将得到通知。在哪里调用 **setChanged()** 取决于程序的逻辑设计。

现在我们进入了一个进退两难的窘境。被观察的对象将有不只一个类似的可选择项。例如，假设有一个用于处理 GUI 的可选择项——比如按钮——类似的可选择项有鼠标击发按钮、鼠标在按钮上方移过以及（由于某些原因）改变按钮颜色等。所以我们希望能够向不同的观察者报告所有这些事件，而每个观察者只对一种不同类型的事件感兴趣。

问题是，在这种情况下要达到以上目的采用多重继承：“为了处理鼠标击发按钮从 **Observable** 中继承，为了处理鼠标在按钮上方移动从 **Observable** 中继承，如此等等，好啦，…哦！这不可能实现。”

### 10.13.1 “内部类”方法

在某些情况下，必须（有效地）向上类型转换（upcast）成为多个不同的类型，但是在这种情况下，需要为同一个基类型提供几个不同的实现。从 Java 中引进了这种解决方法，这种方法比 C++ 的嵌套类更优越。Java 有一个被称为内部类的内建特征，它很像 C++ 中的嵌套类，但是它能够通过隐式使用内部类创建的对象“this”指针来访问其包含（外围）类的非静态数据成员。<sup>①</sup>

为了在 C++ 中实现内部类（inner class）方法，必须显式获得和使用指向包含对象的指针。举例如下：

```
//: C10:InnerClassIdiom.cpp
// Example of the "inner class" idiom.
#include <iostream>
#include <string>
using namespace std;
class Poingable {
public:
    virtual void poing() = 0;
};

void callPoing(Poingable& p) {
```

① 内部类和子程序闭包（subroutine closure）有些相似，子程序闭包用于引用一个函数调用的环境以便稍后复制。

```

    p.poing();
}

class Bingable {
public:
    virtual void bing() = 0;
};

void callBing(Bingable& b) {
    b.bing();
}

class Outer {
    string name;
    // Define one inner class:
    class Inner1;
    friend class Outer::Inner1;
    class Inner1 : public Poingable {
        Outer* parent;
    public:
        Inner1(Outer* p) : parent(p) {}
        void poing() {
            cout << "poing called for "
                  << parent->name << endl;
            // Accesses data in the outer class object
        }
    } inner1;
    // Define a second inner class:
    class Inner2;
    friend class Outer::Inner2;
    class Inner2 : public Bingable {
        Outer* parent;
    public:
        Inner2(Outer* p) : parent(p) {}
        void bing() {
            cout << "bing called for "
                  << parent->name << endl;
        }
    } inner2;
public:
    Outer(const string& nm)
        : name(nm), inner1(this), inner2(this) {}
    // Return reference to interfaces
    // implemented by the inner classes:
    operator Poingable&() { return inner1; }
    operator Bingable&() { return inner2; }
};

int main() {
    Outer x("Ping Pong");
    // Like upcasting to multiple base types!:
    callPoing(x);
    callBing(x);
} ///:~

```

这个例子（有意以最简单的语法形式来说明这种方法；在后面很快就可以看到其实际的应用）以接口**Poingable**和**Bingable**开始，每个接口包含一个成员函数。由**callPoing()**和**callBing()**提供的服务要求它们接收的对象分别实现相应的**Poingable**和**Bingable**接口，除此之外它们对对象没有别的请求，这就使得使用**callPoing()**和**callBing()**具有最大限度的灵活性。注意，这两个接口中都缺少**virtual**析构函数——这就意味着不能通过接口来完成

析构对象。

类**Outer**的构造函数包含一些私有数据（如**name**），它希望同时提供**Poingable**和**Bingable**两个接口，这样它就能同**callPoing()**和**callBing()**一起使用。（在这种情形下也可以仅使用多重继承，但在这里为清晰起见而保持简单的程序结构。）为了能够在**Outer**不派生自**Poingable**的前提下提供**Poingable**对象，这里使用了内部类方法。首先，**class Inner**的声明说明这是一个名为**Inner**的嵌套类。这就允许在后面能够将其声明为外部类**Outer**的友元类。其次，现在嵌套类能够访问外部类**Outer**的所有私有成员，现在就可以定义嵌套类了。注意，嵌套类有一个指向用于创建**Outer**对象的指针，这个指针必须在嵌套类的构造函数中进行初始化。最后，来自**Poingable**的**poing()**函数得以实现。另外一个用来实现**Bingable**的内部类采用同样的过程实现。每个内部类只有一个**private**实例被创建，该实例在**Outer**的构造函数中被初始化。通过创建成员对象并返回对它们的引用，排除了对象生存期可能产生的问题。

注意，两个内部类都是**private**的，事实上客户代码都不能访问其任何实现细节，因为两个访问函数**operator Poingable&()**和**operator Bingable&()**只返回一个用来向上类型转换为接口的引用，而不是实现它的对象。事实上，因为两个内部类是**private**的，客户代码甚至不能向下类型转换为实现类，这样就在接口和实现之间提供了完全的隔离。

这里获得了定义自动类型转换函数**operator Poingable&()**和**operator Bingable&()**的额外特权。在**main()**函数中，可以看到这些允许提供一种使得**Outer**看起来像是从**Poingable**和**Bingable**多重继承来的语法形式。不同之处在于这种“类型转换”在此情况下是单向的。只可以得到向上类型转换为**Poingable**或**Bingable**的效果，但是不能向下类型转换回**Outer**。在下面**observer**的例子中，可以看到更典型的方法：通过提供使用普通的成员函数而不是自动类型转换函数来访问内部类对象。

### 10.13.2 观察者模式举例

具备了**Observer**和**Observable**头文件和内部类方法的知识，现在请看一个观察者模式的程序例子：

```
//: C10:ObservedFlower.cpp
// Demonstration of "observer" pattern.
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
#include "Observable.h"
using namespace std;

class Flower {
    bool isOpen;
public:
    Flower() : isOpen(false),
        openNotifier(this), closeNotifier(this) {}
    void open() { // Opens its petals
        isOpen = true;
        openNotifier.notifyObservers();
        closeNotifier.open();
    }
    void close() { // Closes its petals
        isOpen = false;
        closeNotifier.notifyObservers();
        openNotifier.close();
    }
};
```



```

    }
    // Using the "inner class" idiom:
    class OpenNotifier;
    friend class Flower::OpenNotifier;
    class OpenNotifier : public Observable {
        Flower* parent;
        bool alreadyOpen;
    public:
        OpenNotifier(Flower* f) : parent(f),
            alreadyOpen(false) {}
        void notifyObservers(Argument* arg = 0) {
            if(parent->isOpen && !alreadyOpen) {
                setChanged();
                Observable::notifyObservers();
                alreadyOpen = true;
            }
        }
        void close() { alreadyOpen = false; }
    } openNotifier;
    class CloseNotifier;
    friend class Flower::CloseNotifier;
    class CloseNotifier : public Observable {
        Flower* parent;
        bool alreadyClosed;
    public:
        CloseNotifier(Flower* f) : parent(f),
            alreadyClosed(false) {}
        void notifyObservers(Argument* arg = 0) {
            if(!parent->isOpen && !alreadyClosed) {
                setChanged();
                Observable::notifyObservers();
                alreadyClosed = true;
            }
        }
        void open() { alreadyClosed = false; }
    } closeNotifier;
};

class Bee {
    string name;
    // An "inner class" for observing openings:
    class OpenObserver;
    friend class Bee::OpenObserver;
    class OpenObserver : public Observer {
        Bee* parent;
    public:
        OpenObserver(Bee* b) : parent(b) {}
        void update(Observable*, Argument *) {
            cout << "Bee " << parent->name
                << "'s breakfast time!" << endl;
        }
    } openObsrv;
    // Another "inner class" for closings:
    class CloseObserver;
    friend class Bee::CloseObserver;
    class CloseObserver : public Observer {
        Bee* parent;
    public:
        CloseObserver(Bee* b) : parent(b) {}
        void update(Observable*, Argument *) {
            cout << "Bee " << parent->name
                << "'s bed time!" << endl;
        }
    }
};

```



```

    } closeObsrv;
public:
    Bee(string nm) : name(nm),
        openObsrv(this), closeObsrv(this) {}
    Observer& openObserver() { return openObsrv; }
    Observer& closeObserver() { return closeObsrv; }
};

class Hummingbird {
    string name;
    class OpenObserver;
    friend class Hummingbird::OpenObserver;
    class OpenObserver : public Observer {
        Hummingbird* parent;
    public:
        OpenObserver(Hummingbird* h) : parent(h) {}
        void update(Observable*, Argument *) {
            cout << "Hummingbird " << parent->name
                << "'s breakfast time!" << endl;
        }
    } openObsrv;
    class CloseObserver;
    friend class Hummingbird::CloseObserver;
    class CloseObserver : public Observer {
        Hummingbird* parent;
    public:
        CloseObserver(Hummingbird* h) : parent(h) {}
        void update(Observable*, Argument *) {
            cout << "Hummingbird " << parent->name
                << "'s bed time!" << endl;
        }
    } closeObsrv;
public:
    Hummingbird(string nm) : name(nm),
        openObsrv(this), closeObsrv(this) {}
    Observer& openObserver() { return openObsrv; }
    Observer& closeObserver() { return closeObsrv; }
};

int main() {
    Flower f;
    Bee ba("A"), bb("B");
    Hummingbird ha("A"), hb("B");
    f.openNotifier.addObserver(ha.openObserver());
    f.openNotifier.addObserver(hb.openObserver());
    f.openNotifier.addObserver(ba.openObserver());
    f.openNotifier.addObserver(bb.openObserver());
    f.closeNotifier.addObserver(ha.closeObserver());
    f.closeNotifier.addObserver(hb.closeObserver());
    f.closeNotifier.addObserver(ba.closeObserver());
    f.closeNotifier.addObserver(bb.closeObserver());
    // Hummingbird B decides to sleep in:
    f.openNotifier.deleteObserver(hb.openObserver());
    // Something changes that interests observers:
    f.open();
    f.open(); // It's already open, no change.
    // Bee A doesn't want to go to bed:
    f.closeNotifier.deleteObserver(
        ba.closeObserver());
    f.close();
    f.close(); // It's already closed; no change
    f.openNotifier.deleteObservers();
    f.open();
    f.close();
} ///:~

```



在这里，令人感兴趣的事件是**Flower**的打开或关闭。由于内部类方法的使用，这两个事件成为可以独立进行观察的现象。类**OpenNotifier**和**CloseNotifier**都派生自**Observable**，因此它们能够访问**setChanged()**，并且能够处理需要**Observable**的任何事件。请注意，与**InnerClassIdiom.cpp**相反，**Observable**的派生是**public**的。这是因为它们的一些成员函数要求必须能够被客户程序员访问。没有任何规定要求内部类必须为**private**；在**InnerClassIdiom.cpp**中只是遵从“尽可能声明为私有”的设计原则。可以将这些类声明为**private**，并在**Flower**中设定代理来开放那些成员函数，但这并不会有多大好处。

在**Bee**和**Hummingbird**中内部类方法也很便利地定义了多种**Observer**，因为这两个类都需要独立观察**Flower**的打开与关闭。请注意，内部类方法是如何提供了许多和继承一样有益的特性（例如，能够访问外部类中的私有数据）。

在**main()**中，可以看到观察者模式的主要有益之处：以**Observable**动态地注册和注销**Observer**获得在程序运行时改变行为的能力。这个灵活性是以显著增加代码的代价而达到的——读者可能经常能看到在设计模式中的这种折中：增加某处的复杂性以换取另一处的灵活性的提升和（或）复杂性的降低。

如果仔细研究前面的例子，就会发现**OpenNotifier**和**CloseNotifier**使用了基本的**Observable**接口。这意味着，可以从其他完全不同的**Observer**类派生；**Observer**与**Flower**之间惟一的联系是**Observer**接口。

另外一种完成这种细微粒度的可观察现象的方法是对该现象使用某种形式的标记，例如空类、字符串或枚举等表示不同类型的可观察行为。这种方法可以使用聚合而不是继承来实现，不同之处主要在于时间与空间效率间的折中。而对于客户来说，这种差异是可以忽略的。

## 10.14 多重派遣

在处理多个类交互作用的情况时，程序会变得特别散乱。例如，考虑一个解析和执行数学表达式的系统。在系统中希望使用**Number + Number**、**Number \* Number**等方式表达，其中**Number**是一族数值对象的基类。但是如果给出**a + b**，并且不知道**a**或**b**的准确的类型，那么怎样才能让这二者适当地进行交互作用呢？

刚开始回答时，有一些事情可能没有考虑：C++只执行单重派遣（single dispatching）。这就是说，如果在多个不知道类型的对象之间操作，C++只能在其中一个类型上激发动态绑定机制。这不能解决这里描述的问题，因此程序员只能手工发现一些类型并且有效地制造自己的动态绑定行为。

这种解决方法被称为多重派遣（Multiple dispatching）（GoF在访问者模式的语境下描述了这种方法，访问者模式将在下节介绍）。这里只有两个派遣，被称为双重派遣（double dispatching）。读者可能会记得，多态只能通过虚函数调用来实现，所以如果想要发生多重派遣，必须有一个虚函数调用以确定每个未知的类型。因此，如果处理的是不同层次结构的两个类型的交互作用，则每个层次结构都必须有一个虚函数调用。通常，将设立这样一种结构，使得一个成员函数的调用导致多个虚函数调用，并且因此在该过程中确定多个类型：对于每个派遣都需要一个虚函数调用。下面例子中被调用的虚函数是**compete()**和**eval()**，二者都是同一类型的成员函数（对于多重派遣这并不是必要条件）：<sup>①</sup>

① 这个例子出现在其他作者的书籍中之前，用C++和Java两种语言描述的这个例子已在网站www.MindView.net存在了多年而没有归属。



```

//: C10:PaperScissorsRock.cpp
// Demonstration of multiple dispatching.
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>
#include <ctime>
#include <cstdlib>
#include "../purge.h"
using namespace std;

class Paper;
class Scissors;
class Rock;

enum Outcome { WIN, LOSE, DRAW };

ostream& operator<<(ostream& os, const Outcome out) {
    switch(out) {
        default:
            case WIN: return os << "win";
            case LOSE: return os << "lose";
            case DRAW: return os << "draw";
    }
}

class Item {
public:
    virtual Outcome compete(const Item*) = 0;
    virtual Outcome eval(const Paper*) const = 0;
    virtual Outcome eval(const Scissors*) const = 0;
    virtual Outcome eval(const Rock*) const = 0;
    virtual ostream& print(ostream& os) const = 0;
    virtual ~Item() {}
    friend ostream& operator<<(ostream& os, const Item* it) {
        return it->print(os);
    }
};

class Paper : public Item {
public:
    Outcome compete(const Item* it) { return it->eval(this); }
    Outcome eval(const Paper*) const { return DRAW; }
    Outcome eval(const Scissors*) const { return WIN; }
    Outcome eval(const Rock*) const { return LOSE; }
    ostream& print(ostream& os) const {
        return os << "Paper ";
    }
};

class Scissors : public Item {
public:
    Outcome compete(const Item* it) { return it->eval(this); }
    Outcome eval(const Paper*) const { return LOSE; }
    Outcome eval(const Scissors*) const { return DRAW; }
    Outcome eval(const Rock*) const { return WIN; }
    ostream& print(ostream& os) const {
        return os << "Scissors";
    }
};

class Rock : public Item {
public:

```

```

    Outcome compete(const Item* it) { return it->eval(this); }
    Outcome eval(const Paper*) const { return WIN; }
    Outcome eval(const Scissors*) const { return LOSE; }
    Outcome eval(const Rock*) const { return DRAW; }
    ostream& print(ostream& os) const {
        return os << "Rock    ";
    }
};

struct ItemGen {
    Item* operator()() {
        switch(rand() % 3) {
            default:
            case 0: return new Scissors;
            case 1: return new Paper;
            case 2: return new Rock;
        }
    }
};

struct Compete {
    Outcome operator()(Item* a, Item* b) {
        cout << a << "\t" << b << "\t";
        return a->compete(b);
    }
};

int main() {
    srand(time(0)); // Seed the random number generator
    const int sz = 20;
    vector<Item*> v(sz*2);
    generate(v.begin(), v.end(), ItemGen());
    transform(v.begin(), v.begin() + sz,
        v.begin() + sz,
        ostream_iterator<Outcome>(cout, "\n"),
        Compete());
    purge(v);
} ///:~

```

**Outcome**将函数**compete()**返回的不同结果进行分类，**operator<<**简化了显示特定**Outcome**的过程。

**Item**是将被多重派遣的那些类型的基类。**Compete::operator()**有两个**Item\***类型的参数（并不知道两者的确切类型），并且调用**virtual Item::compete()**函数开始双重派遣过程。虚拟机制决定了**a**的类型，因此它激发了在函数**compete()**内部的**a**的具体类型的产生。在保留该类型的基础之上，函数**compete()**调用**eval()**执行第2次派遣。将其自身（**this**指针）作为一个参数传递给函数**eval()**，从而产生一个对重载的**eval()**函数的调用，因此保存了第1次派遣的类型信息。在完成第2次派遣时，两个**Item**对象的确切类型就都知道了。

在**main()**函数中，STL算法**generate()**生成**vector v**中的元素内容，然后**transform()**在两个范围上应用**Compete::operator()**。这个版本的**transform()**产生第1个范围的起始和末尾点（包含双重派遣中使用的左边**Item**）；第2个范围的起始点，这个范围持有从双重派遣中所使用的右边的**Item**；目标迭代器在这个例子中是标准输出；以及用于为每个对象调用的函数对象（一个临时的**Compete**类型）。

建立多重派遣需要做许多工作，但是请记住这样做的好处是在调用的时候能够以简洁的句法表达方式达到预期的效果——而不是编写出笨拙的代码在调用的时候决定一个或多个对象的类型，可以说：“你们两个！不管是什么类型，彼此之间可以适当地进行交互作用。”然而，在

编写多重派遣的程序代码之前，确保这种简洁性是非常重要的。

注意，利用表查找来进行多重派遣是有效的。在这里使用虚函数来进行查找，用来代替进行杂乱的表查找。如果有较多的派遣（并且有增加和修改的可能），表查找也许是更好的解决问题的方法。

### 用访问者模式进行多重派遣

访问者模式（Visitor，GoF中最后一个也是最复杂的一个模式）的目标是将类继承层次结构上的操作与这个层次结构本身分开。这是一个相当古怪的动机，因为在面向对象编程中所做的大部分工作是将数据和操作组合在一起形成对象，并利用多态性根据对象的确切类型自动选择操作的正确变化。

利用访问者模式将操作从类的继承层次结构中提取出来置入一个独立的外部层次结构。“主层次结构”包含一个函数**visit()**，该函数接受任何来自操作层次结构的对象。结果得到了两个类继承层次结构而不是一个。此外，可以看到，“主层次结构”变得很脆弱——如果要增加一个新类，也要强制改动第2个层次结构。因此，GoF认为主层次结构应该“很少地变化”。这个限制非常有限，从而更进一步降低了这种模式的可应用性。

为了便于讨论，假定主类层次结构是固定的；也许它是由其他供应商提供的，不能对该层次结构进行改动。如果有这个库的源代码就可以在基类中增加新的虚函数，但是，由于某些原因这是不可行的。一个更可能的方案就是增加新的虚函数，这样做很笨拙，或者说是难以维护的。GoF主张“在跨越不同的节点类上分配所有这些操作，将导致系统难以理解、维护和修改”。（读者将会看到，这样做将导致访问者模式更加难以理解、维护和修改。）GoF的另外一个主张是，要避免由于使用过多的操作而“玷污”了主层次结构的接口（但是，如果接口太“臃肿”了，应该问一下这个对象要做的事情是否太多了）。

然而，库的创建者必定已预见到，用户将需要加入新的操作到层次结构中去，因此他们将函数**visit()**包含了进去。

因此（假定实际上需要这么做）两难的窘境就是，用户需要向基类中添加新的成员函数，但是由于某种原因用户不能接触到基类。那么该如何处理这种情况呢？

访问者模式建立于前一节内容所示的双重派遣方案之上。访问者模式允许创建一个独立的类层次结构**Visitor**而有效地对主类的接口进行扩展，这个独立的类层次结构将主类上的各种操作“虚化”。主类对象仅“接受”访问者，然后调用访问者的动态绑定的成员函数。因此，创建一个访问者，并将其传递给主层次结构，便可以获得和虚函数一样的效果。举例如下：

```
//: C10:BeeAndFlowers.cpp
// Demonstration of "visitor" pattern.
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
#include <ctime>
#include <cstdlib>
#include "../purge.h"
using namespace std;

class Gladiolus;
class Renuculus;
class Chrysanthemum;

class Visitor {
public:
```



```

        virtual void visit(Gladiolus* f) = 0;
        virtual void visit(Renuculus* f) = 0;
        virtual void visit(Chrysanthemum* f) = 0;
        virtual ~Visitor() {}
    };

    class Flower {
    public:
        virtual void accept(Visitor&) = 0;
        virtual ~Flower() {}
    };

    class Gladiolus : public Flower {
    public:
        virtual void accept(Visitor& v) {
            v.visit(this);
        }
    };

    class Renuculus : public Flower {
    public:
        virtual void accept(Visitor& v) {
            v.visit(this);
        }
    };

    class Chrysanthemum : public Flower {
    public:
        virtual void accept(Visitor& v) {
            v.visit(this);
        }
    };

    // Add the ability to produce a string:
    class StringVal : public Visitor {
    public:
        string s;
    public:
        operator const string&() { return s; }
        virtual void visit(Gladiolus*) {
            s = "Gladiolus";
        }
        virtual void visit(Renuculus*) {
            s = "Renuculus";
        }
        virtual void visit(Chrysanthemum*) {
            s = "Chrysanthemum";
        }
    };

    // Add the ability to do "Bee" activities:
    class Bee : public Visitor {
    public:
        virtual void visit(Gladiolus*) {
            cout << "Bee and Gladiolus" << endl;
        }
        virtual void visit(Renuculus*) {
            cout << "Bee and Renuculus" << endl;
        }
        virtual void visit(Chrysanthemum*) {
            cout << "Bee and Chrysanthemum" << endl;
        }
    };

```



```

struct FlowerGen {
    Flower* operator()() {
        switch(rand() % 3) {
            default:
            case 0: return new Gladiolus;
            case 1: return new Renuculus;
            case 2: return new Chrysanthemum;
        }
    }
};

int main() {
    srand(time(0)); // Seed the random number generator
    vector<Flower*> v(10);
    generate(v.begin(), v.end(), FlowerGen());
    vector<Flower*>::iterator it;
    // It's almost as if I added a virtual function
    // to produce a Flower string representation:
    StringVal sval;
    for(it = v.begin(); it != v.end(); it++) {
        (*it)->accept(sval);
        cout << string(sval) << endl;
    }
    // Perform "Bee" operation on all Flowers:
    Bee bee;
    for(it = v.begin(); it != v.end(); it++)
        (*it)->accept(bee);
    purge(v);
} ///:~

```

**Flower**是主层次结构，**Flower**的各个子类通过函数**accept()**得到一个**Visitor**。**Flower**主层次结构除了函数**accept()**外没有别的操作，因此**Flower**层次结构的所有功能都将包含在**Visitor**层次结构中。注意，**Visitor**类必须要了解**Flower**的所有具体类型，如果添加一个**Flower**的新类型，整个**Visitor**层次结构必须重新工作。

每个**Flower**中的**accept()**函数开始一个双重派遣，如上一节所述的双重派遣。第1次派遣决定了**Flower**的准确类型，第2次派遣决定了**Visitor**的准确类型。一旦知道了它们的准确类型，就可以对这两者执行恰当的操作。

因为其不寻常的动机以及显得愚笨的约束，使得人们极不可能使用访问者模式。GoF的例子是难以令人信服的——首先是编译器（编写编译器的人不是很多，似乎极少有人将访问者模式用于这些编译器中），他们也不适用于其他一些例子，认为用户实际上不可能像这样使用访问者模式来解决问题。为了使用访问者模式，用户将面临比在GoF中表现出来更大的压力从而抛弃普通的面向对象结构——这样做实际上获得了什么益处而值得换来如此多的复杂性和限制呢？当发现需要多个新的虚函数时为何不可以在基类中仅添加它们？或者，如果实际上需要添加新函数到现存的层次结构中而又不能修改那个层次结构，在这种情况下为什么不先考虑尝试使用多重继承呢？（尽管如此，用这种方法“挽救”现存的层次结构的可能性还是很小的。）出于同样的考虑，为了使用访问者模式，现存的层次结构必须一开始就将函数**visit()**包括进来，因为如果它在后面添加进来的话就意味着可以修改这个层次结构，这样就能在该层次结构中添加需要的普通虚函数了。不！访问者模式从开始就必须是体系结构的一部分，为了使用它需要有比在GoF中提到的更伟大的动机。<sup>①</sup>

① 将访问者模式包含在GoF中的动机可能是因为它非常灵巧。在一个专题讨论会上，GoF的一个作者对我们中的一个人这样说过：“访问者是我最喜欢的模式。”

之所以在这里介绍访问者模式，是因为看到它在不该使用的时候被使用了，正如多重继承和任何其他很多方法被不正确地使用一样。在使用访问者模式之前务必三思，多问几个为什么。比如，真的不能在基类中添加新的虚函数了吗？在主层次结构中真的需要限制添加新的类型吗？

## 10.15 小结

正如任何其他抽象的特点，设计模式的特点就是为了使工作更加容易。系统中总是有一些东西在变化——这可能是在软件项目生命周期中代码的变化，或许是在某个程序执行的生命周期期间某些对象的变化。找出变化的东西，利用设计模式封装这些变化，并使这些变化能够得到控制。

人们在进行程序设计时很容易迷恋于使用某个特定的设计模式，并且如果因为刚刚知道如何做就贸然去做也将给自己带来烦恼。最难做到的是什么？有点讽刺意味，是遵循《极限编程》（《Extreme Programming》）中的那句格言：“只要能用，就做最简单的。”仅仅做最简单的东西，不仅能够最快速的实现设计，而且其设计也很容易维护。如果这种最简单的东西不能完成工作，读者很快就会发现，除了花费时间编写复杂的实现方法之外，它们还是不起作用的。

## 10.16 练习

- 10-1 创建程序**SingletonPattern.cpp**的一个变体，使其所有函数成为静态函数。在这种情况下还需要**instance()**函数吗？
- 10-2 基于程序**SingletonPattern.cpp**，创建一个类，此类提供一个与某个服务的连接，这个服务向（从）一个配置文件中存取数据。
- 10-3 基于程序**SingletonPattern.cpp**，创建一个管理其固定数目对象的类。假定这些对象是数据库连接，在任何一次读/写操作中只允许使用这些对象中的某个固定数目的对象。
- 10-4 通过向系统中增加另外一种状态来修改程序**KissingPrincess2.cpp**，这样每次亲吻之后将使**creature**青蛙王子进入下一状态。
- 10-5 在《C++编程思想》第2版第1卷和第2卷中（可以从[www.BruceEckel.com](http://www.BruceEckel.com)下载）找到头文件**C16:TStack.h**。为这个类创建这样一个适配器，使该类能够使用此适配器将**STL** 算法**for\_each()**应用于**TStack**的元素。创建一个元素类型为**string**的**TStack**，用字符串元素填充它，并且使用**for\_each()**对**TStack**中的所有字符串中的所有字母字符进行计数。
- 10-6 创建一个能够从命令行中获取文件名列表的框架（即使用模板方法模式）。它把除了最后一个文件以外的所有文件作为读文件打开，最后一个文件作为写文件打开。该框架使用不确定的方法处理每个输入文件，并且将输出写到最后一个文件。利用继承于自定义的这个框架去创建两个独立的应用程序：
  - 1) 将每个文件中的所有字母转换为大写。
  - 2) 在这些文件中寻找由第1个文件给出的那些单词。
- 10-7 使用策略模式而不是模板方法模式来修改练习10-6。
- 10-8 修改程序**Strategy.cpp**使其包含状态行为，使其在对象**Context**的生存期期间能够改变策略。
- 10-9 修改程序**Strategy.cpp**，使用职责链方法，使其能尝试从不同方法中选取一种来显示出它们的名字并且不容许忘记它。
- 10-10 在程序**ShapeFactory1.cpp**中增加一个**Triangle**类。
- 10-11 在程序**ShapeFactory2.cpp**中增加一个**Triangle**类。

- 10-12 在程序**AbstractFactory.cpp**中增加一个名为**GnomesAndFairies**的**GameEnvironment**新类型。
- 10-13 修改程序**ShapeFactory2.cpp**让它用一个抽象工厂模式来创建不同的图形集合（比如说，一个特定的工厂类型对象创建“厚图形”，另一工厂类型对象创建“薄图形”，但是每个工厂对象都能够创建所有图形：圆形、矩形、三角形等等）。
- 10-14 修改程序**VirtualConstructor.cpp**使其能够在**Shape::Shape(string type)**中使用**map**而不是**if-else**语句。
- 10-15 将一个文本文件分解成单词的一个输入流（为简洁起见：输入流在空白字符处进行分解）。创建一个能将单词送入一个集合**set**里的构建器（builder），和另外一个能生成包含单词和统计单词出现次数的**map**的构建器（即对单词进行计数）。
- 10-16 在两个类中创建一个最小限度的**Observer-Observable**设计，在这个设计中没有基类，在文件**Observer.h**中没有额外的参数，并且在文件**Observable.h**中没有成员函数。仅仅用这两个类创建这个最小限度的设计，然后创建一个**Observable**和多个**Observer**，并使**Observable**更新**Observer**来演示你的设计。
- 10-17 修改程序**InnerClassIdiom.cpp**使**Outer**用多重继承而非内部类方法来实现。
- 10-18 修改程序**PaperScissorsRock.java**，使用表查找而非双重派遣。最容易的方法就是创建一个**map**的**map**，将每个对象的每个**map**的**typeid(obj).name()**信息作为其关键字。然后就可以这样查找：**map[typeid(obj1).name()][typeid(obj2).name()]**。注意如何能使系统配置更加简化。何时使用此方法比使用硬编码动态派遣更合适？能否创建一个不使用表查找而使用动态派遣的句法简洁的系统？
- 10-19 创建一个商业模型环境，其拥有3个**Inhabitant**的类型：**Dwarf**（用于工程师）、**Elf**（用于销售人员）以及**Troll**（用于经理）。现在创建一个名为**Project**的类，该类可以实例化不同的人，并且使用多重派遣使他们在相互之间使用**interact()**。
- 10-20 修改练习10-19，以便使其交互作用更加细化。每个**Inhabitant**能够利用**getWeapon()**随机产生一个**Weapon**：**Dwarf**使用**Jargon**或**Play**，**Elf**使用**InventFeature**或**SellImaginaryProduct**，**Troll**使用**Edict**和**Schedule**。在每次交互作用中决定哪些武器“赢”或“输”（就像在文件**PaperScissorsRock.cpp**中一样）。在**Project**中增加一个成员函数**battle()**，这个函数获得两个**Inhabitant**并且使它们进行对抗比赛。现在为**Project**创建一个成员函数**meeting()**，该成员函数用于创建**Dwarf**组、**Elf**组和**Manager**组，并且用函数**battle()**让这几个小组相互之间对抗，直到只有一个小组的成员剩下。这个小组就是胜利者。
- 10-21 在程序**BeeAndFlowers.cpp**加入一个**Hummingbird Visitor**。
- 10-22 加入一个**Sunflower**类型到程序**BeeAndFlowers.cpp**中，注意这样原程序需要怎样修改才能适应新增的类型？
- 10-23 修改程序**BeeAndFlowers.cpp**，不使用访问者模式，而“恢复”使用平常的类层次结构。并且将**Bee**变成一个收集参数方法。

# 并 发

对象提供了将一个程序分解为若干个独立部分的途径。在实际的工作中也经常需要把一个程序分割成若干个分开的、独立运行的子任务。

使用多线程处理 (multithreading), 每个独立的子任务都会被执行的线程 (thread of execution) 驱动, 程序就好像每个线程都拥有自己的CPU。其底层实现机制实际上为线程划分出了CPU时间, 但是在一般情况下, 程序员在编程时并不需要去考虑它, 这有助于简化多线程编程。

进程 (process) 是在其自己的地址空间运行的自含式 (self-contained) 程序。周期性地把CPU从一个任务切换到另一个任务, 多任务处理 (multitasking) 操作系统在同一时刻可以运行多个进程 (程序), 使得它们看上去就好像都在独自运行。线程 (thread) 是一个进程内的单一连续的控制流。因此一个进程可以有多个并发执行的线程。由于这些线程运行在一个进程内, 所以它们分享内存和其他资源。编写多线程处理程序中主要的困难就是在不同线程之间协调对这些资源的使用。

多线程处理有多种应用, 而当程序的某些部分与一个特定事件或资源结合在一起的时候, 最经常需要使用多线程。为了防止挂起程序的其余部分, 需要创建一个与那个特定事件或资源关联在一起的线程, 并使这个线程独立于主程序运行。

学习并发编程像是步入了一个崭新的世界, 类似学习一门新的编程语言, 或至少是学习一组新的语言概念。随着在大多数的微机操作系统中出现了支持线程的操作, 在编程语言或者程序库中, 也出现了用于线程的功能扩充。总而言之, 线程编程:

- 1) 不仅看起来神秘, 而且需要人们转换一下思考编程的方式。
- 2) 各种语言中对线程的支持看上去都是相似的。当理解了线程时, 就会理解一个共同的表述方式。

理解并发编程与理解多态性有类似的难度。经过一番努力, 就可以彻底了解其基本机制, 但一般需要深入地学习和理解才能够真正掌握其实质。本章的目标是给读者打下有关并发编程基本原理的坚实基础, 这样就会理解基本概念并且编写出合理的多线程处理程序。不过读者也要意识到, 这也许会使得你很容易变得太过自信。如果要编写任何复杂的程序, 则需要研读关于这个主题的专著。

## 11.1 动机

使用并发的最能激发人们兴趣的理由之一, 就是产生一个可做出响应的用户界面。考虑一个程序, 其在执行某项强烈需要CPU的操作时, 往往会忽略用户的输入并且无法做出响应。程序既需要继续执行其操作, 又需要把控制权归还给用户界面, 这样程序才可以响应用户的请求, 这就是问题的关键。如果有一个“退出”按钮, 我们不希望被迫在程序中的每个代码块中轮流检测它的状态。(这将会使数个退出按钮代码贯穿整个程序, 对它的维护很让人头痛。) 然而, 却希望对这些退出按钮能够做出响应, 就好像系统在定期地检测它一样。

传统的函数不可能在继续进行其操作的同时, 又把控制权归还给程序的其余部分。事实上, 这听起来像是一个不可能完成的任务, 就好像一个CPU必须能同时出现在两个地方, 但这正是



严谨的并发机制提供的错觉效果（在多处理器系统的情形中，这可不只是错觉）。

也可以使用并发机制来优化信息的吞吐量。比如，程序在等待信息输入到达I/O端口的時候可以做些其他重要的工作。要是没有线程处理，惟一可行的解决方法就是不断轮询I/O端口，但这个方法不仅笨拙而且实现起来比较困难。

如果有一台多处理器的计算机（multiprocessor machine），多个线程就可以分布在多个处理器上，用此方法可以极大地提高信息的吞吐量。这种情况通常出现在使用功能强大的多处理器的web服务器上，这样一来，就可以在程序中给每个请求分配一个线程，将大量的用户请求分配到多个CPU来进行处理。

在单CPU计算机上，一个使用多线程的程序仍然一次只能做一件事情，所以不使用任何线程编写出具有相同功能的程序在理论上是可能的。然而，多线程处理提供的重要好处是在程序的组织方面，可以使程序的设计极大的简化。某些类型的问题，比如模拟——例如，一个视频游戏——如果不支持并发是很难解决的。

线程处理模型为编程方式提供了方便，可以在同一时间内魔术般地简化一个程序中的多个操作：CPU将会轮流给每个线程分配一些CPU时间。<sup>①</sup>每个线程都觉得自己一直在占有CPU，但事实上CPU时间被切成片段分配给所有的线程。运行在多CPU计算机上的程序是个例外。但是，关于线程处理的一个重大好处是可以使人们从这一层次中抽出身来，所以代码不需要知道实际上是运行在单CPU计算机上还是多CPU计算机上。<sup>②</sup>因此，使用线程是创建透明可扩展程序的一条途径——如果一个程序运行得太慢，可以很容易地给所使用的计算机增加CPU来加速程序的运行。现在的趋向是，进行多任务处理和多线程处理是利用多处理器系统最合理的途径。

线程处理多少会降低进行计算的效率，但是从改善程序设计、资源平衡以及给用户方便等方面来说，还是相当值得的。一般情况下，使用线程能够创建一个更加松散耦合的设计（loosely coupled design）；否则，部分代码将被迫对这些通常由线程处理的工作花费更大的精力。

## 11.2 C++中的并发

在C++标准委员会创建最初的C++标准时，并发机制被明确排除在外，因为C没有并发，也还因为有许多极具竞争力的近似方法可以实现并发处理。似乎有太多的限制迫使程序员只能用这些方法中的一个。

然而，那些可供选择的方法，结果被证明是错的。为使用并发，就要找到和学习一个库，这就涉及库的特性和某个特定（软件）供应商的产品在工作时是否可靠的问题。另外，没有人能够保证这样的库能运行在不同的编译器上或者跨不同的平台运行。并且，既然并发不是标准语言的一部分，所以要找到懂得并发编程的C++程序员也会更困难。

另一种有影响的Java语言，把并发包含在核心语言中。尽管多线程处理仍然是复杂的，但Java程序员在学习的起始就注意学习多线程并从一开始就使用它。

C++标准委员会正在考虑把支持并发处理的功能加入到下一代C++语言中，但是在这一次正在编写的新版本中，还不清楚加入并发处理的库看起来像什么样子。因此，作者决定把ZThread库作为这一章的基础。首选该设计的原因，因为它是源代码开放的，并且可以免费在

① 当系统使用时间分片机制时（比如Windows）这是正确的。Solaris 使用一个FIFO 并发模型：除非一个更高优先级的线程被唤醒，当前的线程会一直运行直到它被阻塞或终止。那意味着其他有相同优先级的线程直到当前线程放弃处理器后才会运行。

② 假设我们为多CPU设计了它。否则在一个时间分片的单处理器系统上似乎运行良好的代码在移植到多CPU系统上时会失败，这是由于额外的CPU会引发问题而单CPU系统则不会。

<http://zthread.sourceforge.net> 网站上获取。ZThread的作者, IBM的Eric Crahen, 为本章提供了很多有用的工具。<sup>①</sup>

本章仅使用ZThread库的一个子集, 以传达线程处理的基本思想。值得注意的是, ZThread库还包含重要的比这里所展现的更复杂的对线程的支持, 应该深入研究这个库才能更进一步地、完全理解它的性能。

### 安装ZThread

请注意, ZThread库是一个独立的项目, 本教材的作者并不支持它; 只是在本章中使用这个库, 不能提供关于安装发行 (installation issue) 的技术支持。浏览ZThread的网站可以得到安装支持以及勘误报告。

ZThread库以源代码形式发布。从ZThread网站将其下载后 (版本2.3或更高的版本), 必须首先编译这个库, 然后装配到项目中来使用这个库。

对大多UNIX风格的操作系统 (Linux、SunOS、Cygwin等等), 编译ZThread首选的方法是使用装配脚本 (configure script)。文件解包 (使用**tar**) 后, 仅执行:

```
./configure && make install
```

从ZThread档案文件的主目录开始编译, 在/usr/local目录安装库的一份拷贝。使用这个脚本时可以自定义一些选项, 包括文件的位置。若要了解详细内容, 可以使用下面这个命令:

```
./configure -help
```

ZThread的代码也被组织成能对其他平台和编译器 (比如Borland、Microsoft和Metrowerks) 进行简化编译的形式。为了完成这个工作, 创建一个新项目, 把ZThread档案文件的src目录中的所有.cxx文件加入到文件列表中, 然后进行编译。同时也要确保把档案文件的include目录包含在该项目的头文件搜索路径 (header search path) 下。具体的细节根据编译器的不同而有所不同, 所以需要比较熟悉这些工具包后才能够使用这个选项。

一旦编译成功, 下一步就是创建一个使用这个重新编译好的库的项目。首先, 让编译器知道头文件放置的位置, 以便程序中的**#include**语句能够正常工作。典型地, 要将如下选项加入到工程:

```
-I/path/to/installation/include
```

如果使用装配脚本, 可以选择任意安装路径作为前缀的路径 (默认的情况是在/usr/local)。如果使用build目录中的某个项目文件, 只需将ZThread档案文件的主目录设置为安装路径。

接下来, 需要在项目中加入一个选项, 这会使连接器 (linker) 知道到哪里去寻找库。如果使用装配脚本, 如下所示:

```
-L/path/to/installation/lib -lZThread
```

如果使用一个已提供的项目文件, 那么应该这样做:

```
-L/path/to/installation/Debug ZThread.lib
```

再说一遍, 如果使用装配脚本, 可以选择任意安装路径作为前缀的路径。如果使用已提供的项目文件, 路径就是ZThread档案文件的主目录。

注意, 如果使用Linux或使用Windows下的Cygwin ([www.cygwin.com](http://www.cygwin.com)), 则不需要修改包含路径或库文件路径; 安装进程及默认选项会做好全部工作。

在Linux下, 也许需要把下面的东西添加到 **.bashrc** 文件中, 以便让运行时系统 (runtime

① 本章的大部分开始于《Thinking in Java, 3 third edition》(Prentice Hall 2003) 的“并发”一章, 而在处理上有非常显著的改变。

system) 能够在执行本章中的程序时找到共享库文件 **LibZThread-x.x.so.0**:

```
export LD_LIBRARY_PATH=/usr/local/lib:${LD_LIBRARY_PATH}
```

(假设使用的是默认安装进程和位于路径 `/usr/local/lib` 下的共享库; 否则, 把路径改成用户所使用的位置。)

### 11.3 定义任务

一个线程执行一个任务 (task), 所以需要用某种方法来描述这个任务。**Runnable**类提供了一个公共接口来执行任何任意的任务。在这里, **Runnable**类是ZThread 库的核心, 在安装完ZThread 库后, 可以在include目录下的**Runnable.h**文件中找到它:

```
class Runnable {
public:
    virtual void run() = 0;
    virtual ~Runnable() {}
};
```

把**Runnable**类做成一个抽象基类, **Runnable**类就可以很容易地将一个基类与其他类结合起来。

为了定义一个任务, 可以从**Runnable**类继承并且重载**run()**函数, 使任务去做命令它做的事情。

例如, 下面的这个**LiftOff**任务显示了在火箭发射离地升空前倒计时:

```
///C11:LiftOff.h
// Demonstration of the Runnable interface.
#ifndef LIFTOFF_H
#define LIFTOFF_H
#include <iostream>
#include "zthread/Runnable.h"

class LiftOff : public ZThread::Runnable {
    int countDown;
    int id;
public:
    LiftOff(int count, int ident = 0) :
        countDown(count), id(ident) {}
    ~LiftOff() {
        std::cout << id << " completed" << std::endl;
    }
    void run() {
        while(countDown-->0)
            std::cout << id << ":" << countDown << std::endl;
        std::cout << "Liftoff!" << std::endl;
    }
};
#endif // LIFTOFF_H ///
```

标识符**id**能区别该任务的多个实例。如果只创建了单个实例, 可以使用**ident**的默认值。析构函数允许读者看到一个任务已被正确地销毁。

在下面的例子中, 任务的**run()**函数不是被单独的线程驱动; 它在**main()**函数中仅被直接调用。

```
///C11:NoThread.cpp
#include "LiftOff.h"

int main() {
```

```
LiftOff launch(10);
    launch.run();
} ///:~
```

当一个类从**Runnable**派生出来的时候，它必须有一个**run()**函数，但它却没有什么特别的——没有产生任何天生的线程处理的能力。

为完成线程处理的行为，必须使用线程类**Thread**。

## 11.4 使用线程

为了使用线程驱动**Runnable**对象，就要创建独立的**Thread**对象，并且把一个**Runnable**指针传递给**Thread**的构造函数。这样就完成了线程的初始化，然后调用**Runnable**的 **run()** 函数将其作为一个可中断线程。使用一个**Thread**来驱动**LiftOff**，下面的例子显示了任何任务可以怎样在其他线程的语境中运行。

```
///: C11:BasicThreads.cpp
// The most basic use of the Thread class.
//{L} ZThread
#include <iostream>
#include "LiftOff.h"
#include "zthread/Thread.h"
using namespace ZThread;
using namespace std;

int main() {
    try {
        Thread t(new LiftOff(10));
        cout << "Waiting for LiftOff" << endl;
    } catch(Synchronization_Exception& e) {
        cerr << e.what() << endl;
    }
} ///:~
```

**Synchronization\_Exception**是ZThread库的一部分，并且是所有ZThread异常的基类。如果在启动或正在使用线程的时候有错误发生，将会抛出这个异常。

**Thread**类构造函数仅需要一个指向**Runnable**对象的指针。创建一个**Thread**对象将为线程完成必要的初始化，然后调用**Runnable**的**run()**成员函数启动该任务。即使**Thread**的构造函数有效地调用了—一个长时间运行的函数，这个构造函数也会快速返回。这里已经使一个成员函数有效地调用了**LiftOff::run()**函数，并且那个成员函数还没有执行完，但是由于**LiftOff::run()**函数正在被一个不同的线程执行，所以仍然可以继续**main()**线程中执行其他操作。（这种能力不仅限于**main()**线程——任何线程都可以启动另外的线程。）运行该程序就可以看到这一点。即使**LiftOff::run()**已经被调用，“Waiting for LiftOff”消息也将会在倒数计数完成之前显现。因此，该程序同一时刻运行了两个函数——**LiftOff::run()**和**main()**。

现在可以很容易地添加更多的线程来驱动更多的任务。在这里，可以看到所有的线程如何与其他的线程协调运行：

```
///: C11:MoreBasicThreads.cpp
// Adding more threads.
//{L} ZThread
#include <iostream>
#include "LiftOff.h"
#include "zthread/Thread.h"
using namespace ZThread;
using namespace std;
```

```

int main() {
    const int SZ = 5;
    try {
        for(int i = 0; i < SZ; i++)
            Thread t(new LiftOff(10, i));
        cout << "Waiting for LiftOff" << endl;
    } catch(Synchronization_Exception& e) {
        cerr << e.what() << endl;
    }
} ///:~

```

**LiftOff**构造函数的第2个参数用于标识每个任务。当运行该程序时将会看到，由于线程被不断地换入换出，以至于不同的任务被混合在一起执行。这种交换处理是由线程调度器(scheduler)自动控制的。如果运行的计算机上有多个处理器，线程调度器会在多个处理器间“安然地分配”(quietly distribute)线程。

**for**循环起先似乎有一点奇怪，因为**t**在**for**循环内作为局部变量被创建，然后立刻就跳出了作用域并被销毁。这就使它显得可能会立刻失去这个线程本身，但可以从输出看到：线程的确正在运行，直到结尾。这就说明了当创建了一个**Thread**对象的时候，相关联的线程就会在线程处理系统内注册，并保持其处于活动状态。即使基于栈的**Thread**对象被丢弃，线程本身也会继续处于活动状态直到其相关联的任务完成。虽然从C++的观点上来看这也许与直觉相悖，线程的概念偏离了准则：一个线程创建一个单独执行的线程，新创建的线程在函数调用结束后，仍然能够持续执行。这种偏离反映在对象消失之后底层线程的持续执行(the persistence of the underlying thread)上。

#### 11.4.1 创建有响应的用户界面

如前所述，使用线程处理的动机之一就是创建有响应的用户界面。虽然在本教材中没有包括图形用户界面，读者还是可以看到基于控制台的用户界面的简单示例。

下面的例子从一个文件中按行读取数据并把它们打印到控制台上，每行显示完成之后会休眠(sleeping)(挂起(暂停执行)当前线程)一秒钟。(稍后，在本章中将会学习到有关休眠的更多知识。)在这个过程中程序不会检查用户输入，所以用户界面是无响应的。

```

//: C11:UnresponsiveUI.cpp {RunByHand}
// Lack of threading produces an unresponsive UI.
//{L} ZThread
#include <iostream>
#include <fstream>
#include <string>
#include "zthread/Thread.h"
using namespace std;
using namespace ZThread;

int main() {
    cout << "Press <Enter> to quit:" << endl;
    ifstream file("UnresponsiveUI.cpp");
    string line;
    while(getline(file, line)) {
        cout << line << endl;
        Thread::sleep(1000); // Time in milliseconds
    }
    // Read input from the console
    cin.get();
    cout << "Shutting down..." << endl;
} ///:~

```



为使程序能够做出响应，可以在一个单独的线程中执行一个显示文件的任务。然后，主线程可以监视用户输入，这样程序就变成有响应的了：

```

//: C11:ResponsiveUI.cpp {RunByHand}
// Threading for a responsive user interface.
//{L} ZThread
#include <iostream>
#include <fstream>
#include <string>
#include "zthread/Thread.h"
using namespace ZThread;
using namespace std;

class DisplayTask : public Runnable {
    ifstream in;
    string line;
    bool quitFlag;
public:
    DisplayTask(const string& file) : quitFlag(false) {
        in.open(file.c_str());
    }
    ~DisplayTask() { in.close(); }
    void run() {
        while(getline(in, line) && !quitFlag) {
            cout << line << endl;
            Thread::sleep(1000);
        }
    }
    void quit() { quitFlag = true; }
};

int main() {
    try {
        cout << "Press <Enter> to quit:" << endl;
        DisplayTask* dt = new DisplayTask("ResponsiveUI.cpp");
        Thread t(dt);
        cin.get();
        dt->quit();
    } catch(Synchronization_Exception& e) {
        cerr << e.what() << endl;
    }
    cout << "Shutting down..." << endl;
} ///:~

```

现在 **main()** 函数线程可以在按下回车 <Return> 键时立刻做出响应，并调用 **DisplayTask** 类的 **quit()** 函数。

这个例子也表明了各个任务之间需要通信——**main()** 函数线程中的任务需要通知 **DisplayTask** 关闭。由于有一个指向 **DisplayTask** 的指针，你也许会认为只对那个指针调用 **delete** 删除它就可以终止该任务，但这样会使程序变得不可靠。这样做的问题在于：当销毁任务时，这个任务可能正在做某些重要的处理，所以就有可能使程序处于不稳定的状态。在这里，由任务自己来决定什么时候关闭是安全的。做这件事最容易的一个办法是，仅需设置一个布尔标记，简单地变更这个标记来通知任务：现在希望该任务停止下来。当该任务到达一个稳定点时会检查那个标记，然后在从 **run()** 返回之前做好清理现场所需的一切工作。当任务从 **run()** 返回时，**Thread** 对象知道该任务已经完成。

虽然这个程序足够简单，应该不会有任何问题，但是仍然还是有一些与任务间通信有关的小缺点。这将是本章稍后所要讨论的一个重要主题。

### 11.4.2 使用执行器简化工作

使用ZThread的执行器(Executor)可以减少编码的工作量。执行器在客户和任务的执行之间提供了一个间接层；客户不再直接执行任务，而是由一个中间的对象来执行该任务。

在**MoreBasicThreads.cpp**中使用一个**Executor**对象而非显式创建**Thread**对象，可以表示这些操作。一个**LiftOff**对象知道如何运行一个指定的任务；就像命令模式(Command Pattern)，它给出一个函数以供调用执行。一个**Executor**对象知道如何建造合适的语境来执行**Runnable**对象。在下面的例子中，**ThreadedExecutor**为每个任务创建一个线程：

```
//: c11:ThreadedExecutor.cpp
//{L} ZThread
#include <iostream>
#include "zthread/ThreadedExecutor.h"
#include "LiftOff.h"
using namespace ZThread;
using namespace std;

int main() {
    try {
        ThreadedExecutor executor;
        for(int i = 0; i < 5; i++)
            executor.execute(new LiftOff(10, i));
    } catch(Synchronization_Exception& e) {
        cerr << e.what() << endl;
    }
} ///:~
```

注意，在某些情况下可以用单个的**Executor**对象来创建和管理系统中的所有线程。必须把线程处理代码放在一个**try**块中，因为如果出现错误的话**Executor**的**execute()**函数可能会抛出**Synchronization\_Exception**异常。对于任何包含同步对象状态转换（启动线程、获得互斥锁(mutex)、等待某些条件等等）的函数这都是正确的，读者稍后将会在本章中学到这些内容。

一旦**Executor**中的所有任务都完成了，程序就会退出。

在前面的例子中，**ThreadedExecutor**为需要运行的每个任务都创建了一个线程，但是用一个不同类型的**Executor**对象来代替**ThreadedExecutor**对象，就可以容易的改变任务的执行方式。在本章中，使用**ThreadedExecutor**就很好了，但在产生的代码中，由于创建了太多的线程，**ThreadedExecutor**将会导致过多的开销。在这种情况下，可以使用**PoolExecutor**对象来替换**ThreadedExecutor**对象，它使用一个有限的线程集以并行的方式执行提交的任务。

```
//: C11:PoolExecutor.cpp
//{L} ZThread
#include <iostream>
#include "zthread/PoolExecutor.h"
#include "LiftOff.h"
using namespace ZThread;
using namespace std;

int main() {
    try {
        // Constructor argument is minimum number of threads:
        PoolExecutor executor(5);
        for(int i = 0; i < 5; i++)
            executor.execute(new LiftOff(10, i));
    } catch(Synchronization_Exception& e) {
```



```

        cerr << e.what() << endl;
    }
} ///:~

```

使用**PoolExecutor**，可以预先将开销很大的线程分配工作一次做完，在可能的时候重用这些线程。这样做会节省时间，因为不会因不断地为了每个任务都创建一个线程而付出那些开销。并且在一个事件驱动的系统，对于一些需要由线程来处理的事件，可以以很快地方式产生。而这些快速产生的线程可以仅从线程池中取出线程的方式来提供。因为**PoolExecutor**使用的**Thread**对象数量是有限的，所以不能滥用这些可用的资源。因此，尽管本教材将会使用**ThreadedExecutor**类，在产生的代码中还是要考虑使用**PoolExecutor**类。

**ConcurrentExecutor**类就像是一个**PoolExecutor**类，该类有大小固定的一个线程。对于需要在另一个线程中不断运行的任何任务（长期处于活动状态的任务）来说，这个类是很有用的，例如一个监听某个信道套接字连接的任务。对于需要在线程中运行的短任务它也是很方便的，比如，更新本地或远程日志的小任务，或者为事件分派线程等。

如果有多个任务被提交至一个**ConcurrentExecutor**，每个任务都会在下一个任务开始之前执行完成，所有的任务都使用同一个线程。在下面的例子中，将会看到，每个任务按其被提交的顺序执行，并且在下一个任务开始之前执行完成。因此，一个**ConcurrentExecutor**对象串行化（顺序执行）提交给它的任务。

```

//: C11:ConcurrentExecutor.cpp
//{L} ZThread
#include <iostream>
#include "zthread/ConcurrentExecutor.h"
#include "LiftOff.h"
using namespace ZThread;
using namespace std;

int main() {
    try {
        ConcurrentExecutor executor;
        for(int i = 0; i < 5; i++)
            executor.execute(new LiftOff(10, i));
    } catch(Synchronization_Exception& e) {
        cerr << e.what() << endl;
    }
} ///:~

```

就像**ConcurrentExecutor**，**SynchronousExecutor**用于需要同一时刻只运行一个任务的时候，串行代替了并发。不像**ConcurrentExecutor**，**SynchronousExecutor**自己不创建或管理线程。它使用提交任务的线程，因此只会作为同步的焦点（focal point for synchronization）来行动。如果有n个线程向**SynchronousExecutor**提交任务，永远不会一次（同一时刻）运行两个任务。另外，每个任务运行完成后，队列里的下一个任务才会开始执行。

例如，假设现在有许多线程运行着使用文件系统的任务，但正在编写的是可移植的代码，所以不想用**flock()**或其他特定的操作系统调用来加锁一个文件。可以在任何线程中和一个**SynchronousExecutor**一起运行这些任务，来保证在同一时刻只有一个任务在运行。这种运行方式，不需要处理共享资源上的同步问题（而且其间不会冲击文件系统）。一个较好的解决方法，就是对资源的访问采用同步方式进行（将在本章稍后学到这些内容），但是，**SynchronousExecutor**可以跳过对适当合理的某些原型事件进行协调的麻烦。



```

//: C11:SynchronousExecutor.cpp
//{L} ZThread
#include <iostream>
#include "zthread/SynchronousExecutor.h"
#include "LiftOff.h"
using namespace ZThread;
using namespace std;

int main() {
    try {
        SynchronousExecutor executor;
        for(int i = 0; i < 5; i++)
            executor.execute(new LiftOff(10, i));
    } catch(Synchronization_Exception& e) {
        cerr << e.what() << endl;
    }
} ///:~

```

当运行该程序时，将会看到任务以其被提交的顺序执行，每个任务在下一个任务启动前完成。但是看不到有新线程创建——因为在本例中，**main()**线程是提交所有任务的线程，所以每个任务中都用到了它。因为**SynchronousExecutor**主要用于原型处理，在产生的代码中不会大量用到它。

#### 11.4.3 让步

如果知道在**run()**函数中的一次遍历循环（大多数**run()**函数包括一个长期运行的(long-running)循环）期间已经完成了所需要做的工作，就可以给线程调度机制一个暗示，现在已经做完了该做的工作，可以让其他线程使用CPU了。这个暗示（它仅仅是个暗示——不能保证所实现的系统会监听到它）以调用**yield()**函数的形式来表示。

下面，在每次循环后使用让步操作，可以产生一个**LiftOff**示例的修改版本。

```

//: C11:YieldingTask.cpp
// Suggesting when to switch threads with yield().
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
#include "zthread/ThreadedExecutor.h"
using namespace ZThread;
using namespace std;

class YieldingTask : public Runnable {
    int countDown;
    int id;
public:
    YieldingTask(int ident = 0) : countDown(5), id(ident) {}
    ~YieldingTask() {
        cout << id << " completed" << endl;
    }
    friend ostream&
    operator<<(ostream& os, const YieldingTask& yt) {
        return os << "#" << yt.id << ": " << yt.countDown;
    }
    void run() {
        while(true) {
            cout << *this << endl;
            if(--countDown == 0) return;
            Thread::yield();
        }
    }
};

```



```

int main() {
    try {
        ThreadedExecutor executor;
        for(int i = 0; i < 5; i++)
            executor.execute(new YieldingTask(i));
    } catch(Synchronization_Exception& e) {
        cerr << e.what() << endl;
    }
} ///:~

```

可以看到，任务的**run()**成员函数完全由一个无限循环组成。使用**yield()**比不使用它时，程序的输出均衡了许多。可以试着注释掉**Thread::yield()**调用，看看有何不同。然而在一般情况下，**yield()**只在极少的情形下有用处，别想依赖它来对应用程序做出任何严谨的调整。

#### 11.4.4 休眠

可以控制线程行为的另一种办法，就是调用函数**sleep()**，使线程根据给定的毫秒数停止执行一段时间。在前面的例子中，如果用调用**sleep()**而非调用**yield()**，就会得到下面的程序：

```

//: C11:SleepingTask.cpp
// Calling sleep() to pause for awhile.
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
#include "zthread/ThreadedExecutor.h"
using namespace ZThread;
using namespace std;

class SleepingTask : public Runnable {
    int countDown;
    int id;
public:
    SleepingTask(int ident = 0) : countDown(5), id(ident) {}
    ~SleepingTask() {
        cout << id << " completed" << endl;
    }
    friend ostream&
    operator<<(ostream& os, const SleepingTask& st) {
        return os << "#" << st.id << ": " << st.countDown;
    }
    void run() {
        while(true) {
            try {
                cout << *this << endl;
                if(--countDown == 0) return;
                Thread::sleep(100);
            } catch(Interrupted_Exception& e) {
                cerr << e.what() << endl;
            }
        }
    }
};

int main() {
    try {
        ThreadedExecutor executor;
        for(int i = 0; i < 5; i++)
            executor.execute(new SleepingTask(i));
    } catch(Synchronization_Exception& e) {
        cerr << e.what() << endl;
    }
} ///:~

```



**Thread::sleep()**可以抛出一个**Interrupted\_Exception**异常（将在稍后学到中断的概念），可以看到这个异常在**run()**中被捕获。但是该任务是在**main()**函数的**try**块中创建和执行的，这个**try**块捕获**Synchronization\_Exception**（所有**ZThread**异常的基类）异常，因此在**run()**中可能忽略异常并假设异常会传播到**main()**函数中的异常处理器，这种情况有可能发生吗？这种情况不会发生，因为异常不会跨越线程传播倒退回**main()**。因此，必须对可能在任务中出现的任何局部性异常进行处理。

读者会注意到，线程倾向于以任意顺序运行，这意味着**sleep()**也不是一个控制线程执行顺序的办法。它仅会让线程的运行停止片刻。只能保证线程休眠最少100毫秒（在本例中），但线程恢复执行前可能要花更长时间，因为在休眠间歇过期后，线程调度器还需要时间来恢复它。

如果必须要控制线程的执行顺序，最好的办法是使用同步控制（稍后讲述），或者在某些情况下，根本不使用线程，而是自己编写以特定的顺序相互控制的协作子例程（cooperative routine）。

#### 11.4.5 优先权

线程的优先权（priority），向线程调度器传达了一个线程的重要性。虽然CPU以不确定的顺序运行一个线程集，但是在这些等待的线程中，线程调度器将倾向于先运行有最高优先权的等待线程。然而，这并不意味着有较低优先权的线程就不会运行（也就是说，不会因为优先权的问题发生死锁）。有较低优先权的线程只不过趋向于运行较少而已。

这里有一个修改了的**MoreBasicThreads.cpp**，可以用来演示优先权的等级。线程的优先权通过使用**Thread**的**setPriority()**函数来进行调整。

```
//: C11:SimplePriorities.cpp
// Shows the use of thread priorities.
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
using namespace ZThread;
using namespace std;

const double pi = 3.14159265358979323846;
const double e = 2.7182818284590452354;

class SimplePriorities : public Runnable {
    int countDown;
    volatile double d; // No optimization
    int id;
public:
    SimplePriorities(int ident=0): countDown(5), id(ident) {}
    ~SimplePriorities() {
        cout << id << " completed" << endl;
    }
    friend ostream&
    operator<<(ostream& os, const SimplePriorities& sp) {
        return os << "#" << sp.id << " priority: "
            << Thread().getPriority()
            << " count: " << sp.countDown;
    }
    void run() {
        while(true) {
            // An expensive, interruptable operation:
            for(int i = 1; i < 100000; i++)
                d = d + (pi + e) / double(i);
            cout << *this << endl;
            if(--countDown == 0) return;
        }
    }
}
```



```

    }
};

int main() {
    try {
        Thread high(new SimplePriorities);
        high.setPriority(High);
        for(int i = 0; i < 5; i++) {
            Thread low(new SimplePriorities(i));
            low.setPriority(Low);
        }
    } catch(Synchronization_Exception& e) {
        cerr << e.what() << endl;
    }
}
} ///:~

```

在这里，插入符 `operator<<()` 被重载，用来显示任务的标识符、优先权，以及 `countDown` 值。

可以看到，线程 `high` 的优先权处于最高级，其他所有线程被设置为最低优先级。在本例中没有使用 `Executor`，这是因为需要直接访问线程以便设置它们的优先级。

在 `SimplePriorities::run()` 内部，一个开销相当大的浮点计算被重复执行了 100 000 次，包括 `double` 类型的加法和除法。变量 `d` 是 `volatile`（可变的）的，用来确保编译器不对其进行最优化。如果没有这个计算，就不会观察到设置优先级的效果。（可以试一下：注释掉包含 `double` 计算的 `for` 循环。）有了这个计算，就可以观察到 `high` 线程被线程调度器赋优先选择。（至少，这是装有 Windows 操作系统的机器的行为。）计算要持续足够长的时间，使得线程调度机制能够介入，来改变线程和注意它们的优先权，这样就使 `high` 线程得到优先选择。

还可以使用 `getPriority()` 函数获得已有线程的优先权，并且可以在任何时候（不一定非得在线程运行之前，就像在 `SimplePriorities.cpp` 中一样）用 `setPriority()` 函数改变它的优先权。

将优先权映射到操作系统的做法是有问题的。例如，Windows 2000 有 7 个优先级别，而 Sun 的 Solaris 系统有  $2^{31}$  个优先级别。只有将优先级别划分成非常大的粒度才是一个接近实用的方法，就像在 ZThread 库中使用的 `Low`、`Medium` 和 `High` 这样的 3 级优先级的划分。

## 11.5 共享有限资源

可以认为单线程处理程序就像围绕问题空间求解的一个实体，在某一时刻只做一件事情。因为只有一个实体，根本无须考虑在同一时刻两个实体试图使用同一资源的问题：比如两个人试图在同一车位停车，或两个人同时走过同一扇门，甚至两个人同时讲话这样的问题。

有了多线程处理，可以同时做很多事情，但是现在可能有两个或更多的线程试图在同一时刻使用同一个资源。这就可能引起两种不同的问题。首先，必需的资源可能不存在。在 C++ 中，程序员在对象的生存期内对其有完全的控制权。创建线程来使用这些对象是很容易的，这些对象在线程完成之前被销毁。

第 2 个问题是，两个或更多的线程在其试图同时访问同一个资源时可能会发生冲突。如果不去防止这样的冲突，就会有两个线程试图同时访问同一银行账号、在同一打印机上打印、调整同一个变量的值等等问题。

本节介绍当任务仍然在使用某个对象时，而这个对象却突然消失了的问题，以及任务发生冲突时结束共享资源的问题。读者将会学到用来解决这些问题的有关工具。

### 11.5.1 保证对象的存在

在 C++ 中，对内存和资源管理是主要的关注点。在创建任何 C++ 程序时，可以选择在栈上

或者在堆（使用**new**）上创建对象。在一个单线程处理的程序中，通常很容易保持对对象生存期的跟踪，所以不要尝试使用已经销毁的对象。

本章中的示例显示在堆上使用**new**创建了**Runnable**对象，但请注意这些对象从来都不是被显式删除的。然而，当运行程序时，可以从输出中看到，线程库保持跟踪每个任务并最后删除它，这是因为调用了任务的析构函数。这是在**Runnable::run()**成员函数完成时发生的——从**run()**返回就显示任务已经完成。

让线程来负担删除任务是个问题。因为线程不用必须知道是否有另一个线程仍然需要获得对那个**Runnable**的引用，所以可能会提早销毁该**Runnable**。为了处理这个问题，ZThread中的任务被ZThread库机制自动地进行了引用计数（reference-counted）。任务一直维持到该任务的引用计数归零，此时才能够删除该任务。这就意味着，必须总是动态删除任务，所以它们不能在栈上创建。取而代之，任务必须总是用**new**来创建，就像在本章所有例子中看到的那样。

通常必须确保非任务对象在任务需要它们的时候长期保留在活动状态。否则，容易导致那些被任务使用的对象在任务完成之前离开作用域。如果这种情况发生，任务将尝试访问非法的存储单元，并将引起程序错误。这里有一个简单的例子：

```
//: C11:Incrementer.cpp {RunByHand}
// Destroying objects while threads are still
// running will cause serious problems.
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
#include "zthread/ThreadedExecutor.h"
using namespace ZThread;
using namespace std;

class Count {
    enum { SZ = 100 };
    int n[SZ];
public:
    void increment() {
        for(int i = 0; i < SZ; i++)
            n[i]++;
    }
};

class Incrementer : public Runnable {
    Count* count;
public:
    Incrementer(Count* c) : count(c) {}
    void run() {
        for(int n = 100; n > 0; n--) {
            Thread::sleep(250);
            count->increment();
        }
    }
};

int main() {
    cout << "This will cause a segmentation fault!" << endl;
    Count count;
    try {
        Thread t0(new Incrementer(&count));
        Thread t1(new Incrementer(&count));
    } catch(Synchronization_Exception& e) {
        cerr << e.what() << endl;
    }
}
//::~~
```



**Count**类初看上去似乎有点功能过强，但是如果**n**只是一个**int**型变量（不如是一个数组），编译器会把它放在寄存器中，那个存储单元在**Count**对象离开作用域后仍保持可用（虽然从技术上来说这是不合法的）。在这种情况下发现内存违例（violation）是困难的。最终结果依赖使用的编译器和操作系统的不同而有所不同，可以试着使**n**成为一个**int**型变量看看会发生什么。在任何事件中，如果**Count**包含如上的一个**int**数组，编译器被迫要将它放在栈上而非寄存器中。

**Incrementer**是一个使用**Count**对象的简单任务。在**main()**函数中，可以看到**Incrementer**任务运行了足够长的时间，**Count**对象离开了作用域，所以该任务尝试访问一个非长期存在的对象。这就会产生一个程序错误。

为了解决这个问题，必须保证在这些任务之间任何被共享的对象要长期存在，只要这些任务需要它们（如果对象没有被共享，可以把它们直接组成到任务类中，如此一来，使它们的生存期与那个任务捆绑在一起）。既然不希望静态的程序作用域控制对象的生存期，那么就可以把对象放置在堆上。并且确保直到没有其他对象（在此情况下指任务）使用它时才被销毁，这里使用了引用计数。

引用计数在本教材第1卷中有过透彻的讲解，本卷中更进一步地复习它。ZThread库包括一个名叫**CountedPtr**的模板，它自动执行引用计数并在引用计数归零时用**delete**删除一个对象。这里有一个使用**CountedPtr**对上面程序进行了修改的新程序，以防发生这类错误：

```
//: C11:ReferenceCounting.cpp
// A CountedPtr prevents too-early destruction.
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
#include "zthread/CountedPtr.h"
using namespace ZThread;
using namespace std;

class Count {
    enum { SZ = 100 };
    int n[SZ];
public:
    void increment() {
        for(int i = 0; i < SZ; i++)
            n[i]++;
    }
};

class Incrementer : public Runnable {
    CountedPtr<Count> count;
public:
    Incrementer(const CountedPtr<Count>& c) : count(c) {}
    void run() {
        for(int n = 100; n > 0; n--) {
            Thread::sleep(250);
            count->increment();
        }
    }
};

int main() {
    CountedPtr<Count> count(new Count);
    try {
        Thread t0(new Incrementer(count));
        Thread t1(new Incrementer(count));
    } catch(Synchronization_Exception& e) {
        cerr << e.what() << endl;
    }
} ///:~
```



**Incrementer**现在包含一个**CountedPtr**对象，由它管理**Count**。在**main()**函数中，将**CountedPtr**对象以值传递方式传递给两个**Incrementer**对象，所以调用了拷贝构造函数，引用计数增1。只要任务仍然在运行，引用计数值就为非零，所以就不会销毁**CountedPtr**管理的**Count**对象。仅当所有使用**Count**的任务都完成时，**CountedPtr**才会调用（自动地）**Count**对象上的**delete**执行删除操作。

每当有对象被多于一个任务使用时，几乎总是需要使用**CountedPtr**模板来管理那些对象，以防由对象生存期争端而产生的问题。

### 11.5.2 不恰当地访问资源

考虑下面的例子，其中一个任务产生偶数，另外的任务来消费这些数。在这里，消费者线程的惟一工作就是检查偶数的有效性。

首先定义消费者线程**EvenChecker**，因为它在所有后续的例子中会被重复使用。为了使**EvenChecker**与进行试验的各种类型的发生器解除耦合，将创建一个名叫**Generator**的接口，它包含最少量且必需的函数，这些有关的函数是**EvenChecker**必须知道的：它有一个可以取消的**nextValue()**函数。

```
//: C11:EvenChecker.h
#ifndef EVENCHECKER_H
#define EVENCHECKER_H
#include <iostream>
#include "zthread/CountedPtr.h"
#include "zthread/Thread.h"
#include "zthread/Cancelable.h"
#include "zthread/ThreadedExecutor.h"

class Generator : public ZThread::Cancelable {
    bool canceled;
public:
    Generator() : canceled(false) {}
    virtual int nextValue() = 0;
    void cancel() { canceled = true; }
    bool isCanceled() { return canceled; }
};

class EvenChecker : public ZThread::Runnable {
    ZThread::CountedPtr<Generator> generator;
    int id;
public:
    EvenChecker(ZThread::CountedPtr<Generator>& g, int ident)
        : generator(g), id(ident) {}
    ~EvenChecker() {
        std::cout << "~EvenChecker " << id << std::endl;
    }
    void run() {
        while(!generator->isCanceled()) {
            int val = generator->nextValue();
            if(val % 2 != 0) {
                std::cout << val << " not even!" << std::endl;
                generator->cancel(); // Cancels all EvenCheckers
            }
        }
    }
}

// Test any type of generator:
template<typename GenType> static void test(int n = 10) {
    std::cout << "Press Control-C to exit" << std::endl;
    try {
```



```

    ZThread::ThreadedExecutor executor;
    ZThread::CountedPtr<Generator> gp(new GenType);
    for(int i = 0; i < n; i++)
        executor.execute(new EvenChecker(gp, i));
} catch(ZThread::Synchronization_Exception& e) {
    std::cerr << e.what() << std::endl;
}
}
};
#endif // EVENCHECKER_H ///:~

```

**Generator**类引入了抽象类**Cancelable**，它是ZThread库的一部分。**Cancelable**的目的是提供一个一致的接口，以便通过**cancel()**函数来改变对象的状态，用**isCanceled()**函数来检查对象是否已被取消。在这里，使用一个简单的**bool**型取消标志，类似于以前在**ResponsiveUI.cpp**中看到的**quitFlag**。注意，本例中的类是**Cancelable**而不是**Runnable**。另外，依赖于**Cancelable**对象（**Generator**）的所有**EvenChecker**任务都要测试这个标志，看它是否已被取消，就像在**run()**函数中所看到的那样。这种办法，由共享公共资源（**Cancelable Generator**）的任务监视着该资源，以便根据标志来结束监视。这就消除了所谓的竞争条件（race condition），即两个或更多的任务竞争着响应同一个条件，因此发生冲突，否则（没有发生冲突但却）产生不一致的结果。必须仔细考虑以防所有可能会使并发系统崩溃的情形发生。例如，一个任务不能依赖于其他任务，因为不能保证任务停止的顺序。在这里，使任务依赖于非任务对象（使用**CountedPtr**引用计数），消除了潜在的竞争条件。

在稍后各节中，将会看到ZThread库包含与线程结束有关的更通用的机制。

既然多个**EvenChecker**对象可以结束共享一个**Generator**，所以**CountedPtr**模板用于对**Generator**对象进行引用计数。

**EvenChecker**类中的最后一个成员函数是一个**static**静态成员模板。该模板在**CountedPtr**内部创建一个**Generator**，设置和进行对任何类型的**Generator**对象的测试，然后启动若干个使用那个**Generator**的**EvenChecker**。如果**Generator**失败了，**test()**将会报告它并返回；否则，必须按Control-C键来结束它。

**EvenChecker**任务不断地从与其发生联系的**Generator**中读取和测试值。注意，如果**generator->isCanceled()**为真，**run()**函数就返回，它告诉**EvenChecker::test()**中的**Executor**，任务已经完成。任何**EvenChecker**任务可以在与其发生联系的**Generator**上调用**cancel()**函数，这会导致其他所有使用**Generator**的**EvenChecker**顺畅地关闭。

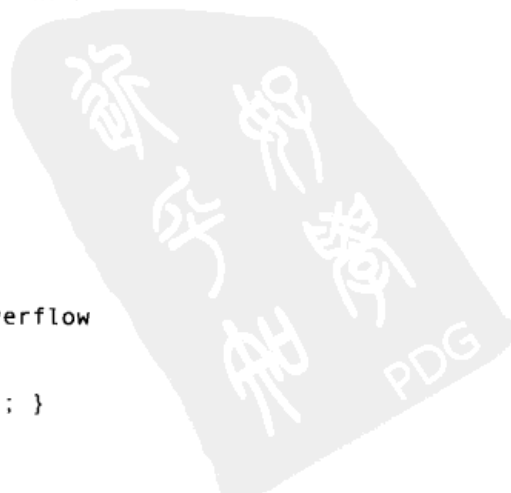
**EvenGenerator**很简单——由**nextValue()**产生下一个偶数值：

```

//: C11:EvenGenerator.cpp
// When threads collide.
//{L} ZThread
#include <iostream>
#include "EvenChecker.h"
#include "zthread/ThreadedExecutor.h"
using namespace ZThread;
using namespace std;

class EvenGenerator : public Generator {
    unsigned int currentEvenValue; // Unsigned can't overflow
public:
    EvenGenerator() { currentEvenValue = 0; }
    ~EvenGenerator() { cout << "~EvenGenerator" << endl; }
    int nextValue() {
        ++currentEvenValue; // Danger point here!
        ++currentEvenValue;
    }
}

```





```

        return currentEvenValue;
    }
};

int main() {
    EvenChecker::test<EvenGenerator>();
} ///:~

```

**currentEvenValue**的值在第1次增1之后与第2次增1之前的这段时间，可能会有一个线程调用**nextValue()**（代码中注释着“Danger point here!”之处），其放进变量的值会处于一个“不正确的”状态。为了证明这种情况可能会发生，**EvenChecker::test()**创建了一组**EvenChecker**对象，不断读取一个**EvenGenerator**的输出，并测试是否每个都为偶数。如果不是，会报告出错并关闭程序。

直到**EvenGenerator**完成多次循环，这个程序可能也不会发现问题，这依赖于你使用的操作系统的特性以及其他实现细节。如果要想尽快地看到它失败，可以尝试把一个**yield()**调用放在第1次与第2次增1操作之间。在任何事件中，当**EvenGenerator**处于“不正确”状态时，因为**EvenChecker**线程仍能够访问**EvenGenerator**里的信息，所以**EvenChecker**最终将会失败。

### 11.5.3 访问控制

前面的例子显示了使用线程时会遇到的一个基本问题：你永远不会知道一个线程何时可能运行。想像一下，你坐在桌子前拿着一把叉子，打算叉盘中最后一块食物。当叉子碰到食物时，它却突然消失了（因为你的线程被挂起，另一个用餐者进来吃掉了食物）。这就是在编写并发程序时要处理的问题。

有时候，在试图使用某一资源时，并不关心它在同一时刻是否正在被访问。但是在大多数情况下还是要关心这个问题。对于多线程处理的工作，需要一些方法来防止两个线程同时访问同一个资源，至少要防止两个线程在临界期（critical period）内访问同一资源。

防止这种冲突的一个简单方法，就是在线程正在使用一个资源时，给该资源加一把锁。访问该资源的第1个线程给资源加上锁，然后其他线程在该资源未被解锁时不能访问它。解锁的同时，另一个要使用它的线程就可以对该资源加锁并且使用它，依此类推。假设汽车的前排座位是有限的资源，那个大喊“我要坐”的小孩就类似于声明获得该锁。

因此，在某个存储单元处于不适当的状态时，需要能够防止任何其他任务访问该存储单元。也就是说，需要有一个机制，当第1个任务已经在使用某个存储单元时，该机制用来排除（exclude）第2个任务对该存储单元的访问。这个想法对所有多线程处理系统来说是基本的，它被称为相互排斥（mutual exclusion）；该机制被简写为互斥（mutex）。ZThread库包含互斥机制，这在**Mutex.h**头文件中进行了声明。

在以上程序中解决这个问题，首先要能够识别临界区（critical section），在临界区中必须应用相互排斥机制；然后，在进入临界区之前获得互斥锁，并在临界区的终点释放（release）它。在任何时刻仅有一个线程可以获得该互斥锁，因此，相互排斥完成。

```

//: C11:MutexEvenGenerator.cpp {RunByHand}
// Preventing thread collisions with mutexes.
//{L} ZThread
#include <iostream>
#include "EvenChecker.h"
#include "zthread/ThreadedExecutor.h"
#include "zthread/Mutex.h"
using namespace ZThread;

```

```

using namespace std;

class MutexEvenGenerator : public Generator {
    unsigned int currentEvenValue;
    Mutex lock;
public:
    MutexEvenGenerator() { currentEvenValue = 0; }
    ~MutexEvenGenerator() {
        cout << "~MutexEvenGenerator" << endl;
    }
    int nextValue() {
        lock.acquire();
        ++currentEvenValue;
        Thread::yield(); // Cause failure faster
        ++currentEvenValue;
        int rval = currentEvenValue;
        lock.release();
        return rval;
    }
};

int main() {
    EvenChecker::test<MutexEvenGenerator>();
} //:~

```

在**MutexEvenGenerator**中增加了一个叫做**lock**的**Mutex**型变量，并且在**nextValue()**函数中使用**acquire()**和**release()**创建了临界区。另外，为了在**currentEvenValue**处于奇数状态时提高语境切换的可能性，一个**Thread::yield()**调用被插入到两个增1语句之间。因为互斥机制防止了多个线程在同一时刻出现在同一个临界区中的情况，所以不会失败。但是如果有可能发生失败，调用**yield()**是促使失败提早发生的很有用的方法。

注意，**nextValue()**函数必须在临界区内部获得返回值，因为如果从临界区中返回，没有释放这个锁，因此将阻止其再次从临界区获得该锁。（这通常会导致死锁（deadlock），在本章的末尾将学到有关这方面的内容。）

第1个进入**nextValue()**的线程获得了锁，那些试图获得该锁的其他任何线程都被阻塞在那里等待，直到第1个线程释放了该锁。这时候，系统的调度机制选择另一个正在等待得到该锁的线程进入**nextValue()**。以这种方法，在同一时刻只有一个线程能通过被互斥锁保护的代码。

#### 11.5.4 使用保护简化编码

当引入异常时，互斥锁的使用就迅速变得复杂起来。为确保互斥锁总能被释放，就必须保证每条可能的异常路径都包含一个对**release()**函数的调用。另外，任何有多条返回路径的函数都必须小心，以保证在合适的地点调用**release()**。

利用下述事实，可以很容易地解决这些问题：基于栈的（自动）对象有一个析构函数，不管是怎样从函数的作用域中退出的，该析构函数总会被调用。在**ZThread**库中，这个功能以**Guard**模板的方式实现。**Guard**模板创建对象，当这些对象被创建时用**acquire()**函数获得一个**Lockable**对象；当这些**Guard**对象被销毁时，用**release()**函数释放该锁。**Guard**对象创建于本地栈上，不管函数是如何退出的，它都将会被自动销毁，并且总能将**Lockable**对象解锁。在这里，把上面的例子用**Guard**重新实现：

```

//: C11:GuardedEvenGenerator.cpp {RunByHand}
// Simplifying mutexes with the Guard template.
//{L} ZThread
#include <iostream>
#include "EvenChecker.h"

```

```

#include "zthread/ThreadedExecutor.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"
using namespace ZThread;
using namespace std;

class GuardedEvenGenerator : public Generator {
    unsigned int currentEvenValue;
    Mutex lock;
public:
    GuardedEvenGenerator() { currentEvenValue = 0; }
    ~GuardedEvenGenerator() {
        cout << "~GuardedEvenGenerator" << endl;
    }
    int nextValue() {
        Guard<Mutex> g(lock);
        ++currentEvenValue;
        Thread::yield();
        ++currentEvenValue;
        return currentEvenValue;
    }
};

int main() {
    EvenChecker::test<GuardedEvenGenerator>();
} ///:~

```

注意，在`nextValue()`函数中，临时返回值不是必须的。一般情况下，要编写的代码较少，因而用户出错的机会大大减少。

**Guard**模板的一个有意思的特征，就是它可以被安全地用于操纵其他保护（guard）。比如，下面程序中的第2个**Guard**可以用于临时解锁一个保护：

```

//: C11:TemporaryUnlocking.cpp
// Temporarily unlocking another guard.
//{L} ZThread
#include "zthread/Thread.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"
using namespace ZThread;

class TemporaryUnlocking {
    Mutex lock;
public:
    void f() {
        Guard<Mutex> g(lock);
        // lock is acquired
        // ...
        {
            Guard<Mutex, UnlockedScope> h(g);
            // lock is released
            // ...
            // lock is acquired
        }
        // ...
        // lock is released
    }
};

int main() {
    TemporaryUnlocking t;
    t.f();
} ///:~

```



**Guard**也可以尝试在一个确定的时间内获得某个锁，然后放弃：

```

//: C11:TimedLocking.cpp
// Limited time locking.
//{L} ZThread
#include "zthread/Thread.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"
using namespace ZThread;

class TimedLocking {
    Mutex lock;
public:
    void f() {
        Guard<Mutex, TimedLockedScope<500>> > g(lock);
        // ...
    }
};

int main() {
    TimedLocking t;
    t.f();
} ///:~

```

在这个例子中，如果在500毫秒内不能获得锁，则抛出一个**Timeout\_Exception**异常。

同步整个类

ZThread库还提供了一个**GuardedClass**模板来自动地为整个类创建同步封装器(wrapper)。这意味着该类中的每个成员函数都将自动被保护。

```

//: C11:SynchronizedClass.cpp {-dmc}
//{L} ZThread
#include "zthread/GuardedClass.h"
using namespace ZThread;

class MyClass {
public:
    void func1() {}
    void func2() {}
};

int main() {
    MyClass a;
    a.func1(); // Not synchronized
    a.func2(); // Not synchronized
    GuardedClass<MyClass> b(new MyClass);
    // Synchronized calls, only one thread at a time allowed:
    b->func1();
    b->func2();
} ///:~

```

对象**a**是非同步的，所以**func1()**和**func2()**能被任意个线程在任何时刻调用。对象**b**被**GuardedClass**封装器保护了起来，所以每个成员函数都被自动同步，在任意时刻每个对象仅有一个函数能被调用。

封装器在类一级的粒度上加锁，这也许会影响到它性能。<sup>①</sup>如果一个类包含某些互不相关的函数，也许用两种不同的锁在内部同步这些函数会更好一些。然而如果这样做了，则意味着

① 这可能很重要。通常函数只有小部分需要被保护。把这些保护放在函数入口点常常可以使临界区比它实际需要的要长。

该类也许包含非强相关 (strongly associated) 的数据组。应该考虑把这个类分解成两个类。

用一个互斥锁保护一个类的所有成员函数并不能自动保证那个类是线程安全 (thread-safe) 的。必须小心考虑所有的线程处理问题, 以便保证线程的安全性。

### 11.5.5 线程本地存储

消除任务在共享资源上发生冲突问题的第2种办法是消除共享变量, 对使用同一对象的各个不同线程, 可以为同一个变量创建不同的存储单元。因此, 如果有5个线程使用一个含有变量 **x** 的对象, 线程本地存储 (thread local storage) 会自动为变量 **x** 产生5个不同的存储片段 (单元)。幸运的是, 线程本地存储的创建和管理由ZThread库的**ThreadLocal**模板自动管理, 如下所示:

```
//: C11:ThreadLocalVariables.cpp {RunByHand}
// Automatically giving each thread its own storage.
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"
#include "zthread/ThreadedExecutor.h"
#include "zthread/Cancelable.h"
#include "zthread/ThreadLocal.h"
#include "zthread/CountedPtr.h"
using namespace ZThread;
using namespace std;

class ThreadLocalVariables : public Cancelable {
    ThreadLocal<int> value;
    bool canceled;
    Mutex lock;
public:
    ThreadLocalVariables() : canceled(false) {
        value.set(0);
    }
    void increment() { value.set(value.get() + 1); }
    int get() { return value.get(); }
    void cancel() {
        Guard<Mutex> g(lock);
        canceled = true;
    }
    bool isCanceled() {
        Guard<Mutex> g(lock);
        return canceled;
    }
};

class Accessor : public Runnable {
    int id;
    CountedPtr<ThreadLocalVariables> tlv;
public:
    Accessor(CountedPtr<ThreadLocalVariables>& tl, int idn)
        : id(idn), tlv(tl) {}
    void run() {
        while(!tlv->isCanceled()) {
            tlv->increment();
            cout << *this << endl;
        }
    }
    friend ostream&
    operator<<(ostream& os, Accessor& a) {
        return os << "#" << a.id << ": " << a.tlv->get();
    }
};
```



```

    }
};

int main() {
    cout << "Press <Enter> to quit" << endl;
    try {
        CountedPtr<ThreadLocalVariables>
            tlv(new ThreadLocalVariables);
        const int SZ = 5;
        ThreadedExecutor executor;
        for(int i = 0; i < SZ; i++)
            executor.execute(new Accessor(tlv, i));
        cin.get();
        tlv->cancel(); // All Accessors will quit
    } catch(Synchronization_Exception& e) {
        cerr << e.what() << endl;
    }
} ///:~

```

当通过实例化该模板来创建**ThreadLocal**对象时，只能用**get()**和**set()**成员函数访问该对象的内容。**get()**函数返回一份与那个线程相关联的对象的拷贝，而**set()**则将其参数插入到与那个线程相关的对象中存储，并返回存储单元中原来所保存的对象。可以看到，这种方法用在了**ThreadLocalVariables**里的**increment()**和**get()**函数中。

由于**tlv**被多个**Accessor**对象共享，它被写成像**Cancelable**一样，以便在想要停止系统运行时，让**Accessor**可以收到信号。

在运行该程序时，将看到各个线程分配有自己的存储单元的证据。

## 11.6 终止任务

在前面的例子中，读者已经看到了使用“退出标志”或**Cancelable**接口以适当的方式来终止一个任务。这是解决该问题的合理的途径。然而，在某些情形下任务却必须要突然地结束。在本节中，读者将会学到有关这样终止任务所产生的后果和存在的问题。

首先，看一个示例，这个示例不仅示范了终止任务的问题，而且也是资源共享的另一个例子。为了介绍这个例子，首先需要解决输入输出流冲突的问题。

### 11.6.1 防止输入/输出流冲突

读者也许已经注意到了前面例子中的输出有时候会出现信息混淆的现象。当初创建C++ 输入/输出流时并没有考虑线程处理的事情，因此没有采取什么措施阻止一个线程的输出与其他线程输出之间的冲突。所以必须编写应用程序来处理输入/输出流同步的问题。

为了解决这个问题，首先需要创建全部的输出数据信息包，然后明确决定什么时候尝试将其发送到控制台。一个简单的解决办法是将信息写入一个**ostream**，然后用一个带有互斥锁的对象作为所有线程的输出点，以防多个线程同时写入数据：

```

//: C11:Display.h
// Prevents ostream collisions.
#ifndef DISPLAY_H
#define DISPLAY_H
#include <iostream>
#include <sstream>
#include "zthread/Mutex.h"
#include "zthread/Guard.h"

class Display { // Share one of these among all threads
    ZThread::Mutex iolock;

```

```

public:
    void output(std::ostream& os) {
        ZThread::Guard<ZThread::Mutex> g(iolock);
        std::cout << os.str();
    }
};
#endif // DISPLAY_H ///:~

```

通过这个办法，预先定义一个标准**operator<<()**函数，可以使用熟悉的**ostream**运算符在内存中构建对象。当一个任务需要显示输出时，它创建一个临时的**ostream**对象，用于构建想要的输出消息。当它调用**output()**时，互斥锁会阻止多个线程同时向该**Display**对象写入数据。（在程序中必须只使用一个**Display**对象，正如在下面例子中将看到的。）

这恰恰表现了其基本思想，但是如果必要的话，可以构建一个更精细的框架。例如，可以把它做成一个单件（Singleton）来强迫实现一个程序中仅有一个**Display**对象的要求。（ZThread库有一个**Singleton**模板用来支持单件。）

### 11.6.2 举例观赏植物园

在这个模拟程序中，公园委员会想要了解每天有多少人通过这个公园的多个入口进入。每个入口有一个十字转门或其他种类的计数器，当该十字转门的计数器增1之后，一个用来表示公园中游客总数的共享计数器也增1。

```

//: C11:OrnamentalGarden.cpp {RunByHand}
//{L} ZThread
#include <vector>
#include <cstdlib>
#include <ctime>
#include "Display.h"
#include "zthread/Thread.h"
#include "zthread/FastMutex.h"
#include "zthread/Guard.h"
#include "zthread/ThreadedExecutor.h"
#include "zthread/CountedPtr.h"
using namespace ZThread;
using namespace std;

class Count : public Cancelable {
    FastMutex lock;
    int count;
    bool paused, canceled;
public:
    Count() : count(0), paused(false), canceled(false) {}
    int increment() {
        // Comment the following line to see counting fail:
        Guard<FastMutex> g(lock);
        int temp = count;
        if(rand() % 2 == 0) // Yield half the time
            Thread::yield();
        return (count = ++temp);
    }
    int value() {
        Guard<FastMutex> g(lock);
        return count;
    }
    void cancel() {
        Guard<FastMutex> g(lock);
        canceled = true;
    }
    bool isCanceled() {

```



```

    Guard<FastMutex> g(lock);
    return canceled;
}
void pause() {
    Guard<FastMutex> g(lock);
    paused = true;
}
bool isPaused() {
    Guard<FastMutex> g(lock);
    return paused;
}
};

class Entrance : public Runnable {
    CountedPtr<Count> count;
    CountedPtr<Display> display;
    int number;
    int id;
    bool waitingForCancel;
public:
    Entrance(CountedPtr<Count>& cnt,
             CountedPtr<Display>& disp, int idn)
        : count(cnt), display(disp), number(0), id(idn),
          waitingForCancel(false) {}
    void run() {
        while(!count->isPaused()) {
            ++number;
            {
                ostringstream os;
                os << *this << " Total: "
                   << count->increment() << endl;
                display->output(os);
            }
            Thread::sleep(100);
        }
        waitingForCancel = true;
        while(!count->isCanceled()) // Hold here...
            Thread::sleep(100);
        ostringstream os;
        os << "Terminating " << *this << endl;
        display->output(os);
    }
    int getValue() {
        while(count->isPaused() && !waitingForCancel)
            Thread::sleep(100);
        return number;
    }
    friend ostream&
    operator<<(ostream& os, const Entrance& e) {
        return os << "Entrance " << e.id << ": " << e.number;
    }
};

int main() {
    srand(time(0)); // Seed the random number generator
    cout << "Press <ENTER> to quit" << endl;
    CountedPtr<Count> count(new Count);
    vector<Entrance*> v;
    CountedPtr<Display> display(new Display);
    const int SZ = 5;
    try {
        ThreadedExecutor executor;
        for(int i = 0; i < SZ; i++) {

```



```

    Entrance* task = new Entrance(count, display, i);
    executor.execute(task);
    // Save the pointer to the task:
    v.push_back(task);
}
cin.get(); // Wait for user to press <Enter>
count->pause(); // Causes tasks to stop counting
int sum = 0;
vector<Entrance*>::iterator it = v.begin();
while(it != v.end()) {
    sum += (*it)->getValue();
    ++it;
}
ostringstream os;
os << "Total: " << count->value() << endl
  << "Sum of Entrances: " << sum << endl;
display->output(os);
count->cancel(); // Causes threads to quit
} catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
}
} //::~~

```

**Count**是一个类，它是用来保存公园游客数的主计数器。单个**Count**对象在**main()**中定义为**count**，同时**count**被作为一个**CountedPtr**实例保存在**Entrance**中，因此被所有**Entrance**对象共享。本例中，使用一个叫**lock**的**FastMutex**模板实例而非普通的**Mutex**，因为**FastMutex**使用本地操作系统的互斥锁并因此产生许多有趣的结果。

在**increment()**函数中，一个使用**lock**对象的**Guard**对象用来同步对**count**的访问。在大约一半时间，这个函数使用**rand()**来引发**yield()**，在这中间取来**count**的数据放入**temp**，并且使**temp**增1，再把**temp**存回到**count**之中。因为这个原因，如果注释掉**Guard**对象的定义，很快就会看到程序崩溃，因为多线程将会同时对**count**进行访问和修改。

**Entrance**类也持有一个本地的**number**，用来记录已通过这个特定入口的游客数。这里提供了对**count**对象的双重检验，以确保所记录的游客数正确。**Entrance::run()**仅使**number**变量和**count**对象增1，并休眠100毫秒。

在主函数中，一个**vector<Entrance\*>**用于装载已经创建的每个**Entrance**。用户按下<Enter>键之后，该**vector**用来迭代所有的个体**Entrance**值并计算其总和。

这个程序在运行时遇到相当少的额外麻烦时，就会以一种稳定的方式关闭所有的对象。编写这个程序的部分原因是为了说明在结束多线程处理程序的执行时需要多么谨慎，还有部分原因是为了示范**interrupt()**函数的值，读者不久就会学到这些。

**Entrance**对象间发生的所有通信都要通过一个**Count**对象。当用户按下<Enter>键时，**main()**函数用**pause()**发送消息给**count**。由于每个**Entrance::run()**都在监视着**count**对象是否暂停下来，这将引发每个**Entrance**对象迁移到**waitingForCancel**等待状态，在这种状态下它将不再计数，但仍然处于活动状态。这是必要的，因为**main()**必须能安全迭代（遍历）在**vector<Entrance\*>**中的每个对象。注意，因为在一个**Entrance**完成计数并迁移至**waitingForCancel**等待状态之前，发生迭代的可能性很小（可以忽略），所以函数**getValue()**循环调用**sleep()**直到对象迁移至**waitingForCancel**等待状态。（这是被称为忙等待（busy wait）的形式之一，是不受欢迎的。稍后会在本章中看到首选的解决办法，它使用了**wait()**函数。）一旦**main()**完成了对**vector<Entrance\*>**的一次遍历迭代，**cancel()**消息就会被送至**count**对象。再强调一次，所有**Entrance**对象都会监视这个状态变化。在这点

上，它们打印一条终止消息并从**run()**中退出，这导致每个任务都会被线程处理机制销毁掉。

当程序运行时，将看到总的计数和当一个游客走过十字转门时每个入口的计数显示。如果注释掉**Count::increment()**中的**Guard**对象，读者就会注意到游客总数不再是预期的值了。每个十字转门所统计的游客数都与**count**中的值不同。只要互斥锁**Mutex**在那里同步对**Counter**的访问，一切就会正常进行。切记：在这里，**Count::increment()**使用**temp**和**yield()**函数放大了失败的潜在可能。在实际的线程处理问题中，从统计学上来说失败的可能性很小，所以读者会很容易陷入相信不会有什么问题会发生的陷阱。就像在上面的例子中，可能会有一些隐藏的问题并没有在这个程序里发生，所以在对并发程序的代码进行复审时应格外仔细。

#### 原子操作

注意，**Count::value()**使用一个**Guard**对象进行同步并返回**count**的值。这就提出一个有趣的观点，因为这段代码不用同步大概也可以在大部分编译器和操作系统上良好运行。其理由就是，在一般情况下一个简单的操作比如返回一个**int**型变量就是一个原子操作（atomic operation），这意味着或许它在一个微处理器指令中完成而不会被中断。（多线程处理机制不能在一个微处理器指令中间停止一个线程。）也就是说，原子操作不能被线程处理机制中断，因此不需要被保护。<sup>①</sup>实际上，如果删除取**count**的值送到**temp**的操作，并且删除**yield()**函数，代之以仅直接**count**增1操作，这样或许不需要进行互斥加锁处理也会工作得很好，因为增1操作通常也是原子操作。<sup>②</sup>

问题在于C++标准并不能保证任何这类操作的原子性。虽然诸如像返回一个**int**型值的操作，和对一个**int**型的值进行增1的操作在大多数计算机上几乎确定地是原子的，但是这并没有保证。正因为没有保证，所以必须考虑最坏的情况。有时可能要调查特定计算机（经常要通过阅读汇编语言）上的原子行为并根据这种假设编写代码。那总归是危险的、欠谨慎的做法。以上相关的信息很容易丢失或者被隐藏，其他人可能会认为这段代码可以被移植到其他机器上，当移植后就会发疯般地追踪由线程冲突而引发的偶然的错误。

所以，虽然从**Count::value()**上删除保护似乎可以照常工作，但并不是无懈可击的，因此可能会在某些机器上看到偏离常规的行为。

### 11.6.3 阻塞时终止

前面例子中的**Entrance::run()**在主循环中包含一个**sleep()**调用。我们知道在那个例子中**sleep()**休眠最后会被唤醒，在任务到达循环的顶部时检查**isPaused()**的状态，便有机会跳出循环。然而，**sleep()**仅是一个线程在其执行过程中被阻塞的一种情况，有时必须终止一个被阻塞的任务。

#### 1. 线程状态

一个线程可以处于以下4种状态之一：

1) 新建 (New) 状态：一个线程只是在被创建的瞬间暂时地保持这个状态。它分配任何必需的系统资源并完成初始化。在这一点它有资格获得CPU时间。线程调度器随后将把该线程转

① 这样说过于简单。有时甚至当它看上去好像是一个原子操作且会是安全的时候它却可能不是，所以当决定不使用同步时必须非常小心。删除用于同步的代码通常是过度优化的一个标志——它会导致我们陷入很多麻烦中而且不会得到更多好处，甚至得不到任何东西。

② 原子性不是惟一的问题。在多处理器系统上，可见性问题比在单处理器上多得多。一个线程所做的改变，即便它们在不能被中断的意义上来说是原子的，对其他线程来说仍然有可能是不可见的（比如，这些改变会被暂时存储在本地处理器缓存中），所以不同的线程会看见应用程序的不同状态。同步机制迫使一个线程做出的改变在多处理器系统上是跨应用程序可见的，然而不使用同步，这些变化何时会变为可见是不确定的。

换到可运行或阻塞状态。

2) 可运行 (Runnable) 状态: 这个状态意味着当时间分片机制为该线程分配可利用的CPU周期时, 线程就可以运行。因此, 在任何时刻, 某个线程可能运行也可能不运行, 但是如果线程调度器安排它, 则没有什么事情会阻止其运行; 这时, 它既不处于死亡状态, 也不处于阻塞状态。

3) 阻塞 (Blocked) 状态: 线程可以运行了, 但有某种事件阻止了它的运行。(比如, 它也许正在等待I/O操作完成。) 当一个线程处于阻塞状态时, 线程调度器会忽略该线程并且不分配给它任何CPU时间。直到线程重新进入可运行状态之前, 它不执行任何操作。

4) 死亡 (Dead) 状态: 一个处于死亡状态的线程, 不能再被调度也不能获得任何CPU时间。它的任务已经完成, 不再是可运行的。使一个线程消逝的正常的办法就是让它从run()函数返回。

## 2. 变为阻塞状态

当一个线程不能继续运行时它就处在阻塞状态。一个线程变为阻塞状态的原因如下:

- 调用sleep(milliseconds)使线程进入休眠状态, 在这种情况下该线程在指定时间内不会运行。
- 已经使用wait()挂起了该线程的运行。在得到signal()或broadcast()消息之前它不会再一次变为可执行状态。我们在后面的小节里将检验这些问题。
- 线程正在等待某个I/O操作完成。
- 线程正在尝试进入一段被一个互斥锁保护的代码块, 而那个互斥锁已经被其他线程获得。

现在需要注意的问题是: 有时需要在某个线程处于阻塞状态时终止它。线程在执行到代码中的某一点上能自己检查状态值并决定结束运行, 如果不能等待线程到达代码中的这一点, 那么就必须强迫线程脱离阻塞状态。

### 11.6.4 中断

正如想象的那样, 在一个Runnable::run()函数的中间跳出, 会比等待函数到达isCanceled()函数的检查点 (或者程序员准备离开函数的其他地方) 时跳出显得更加混乱。当从被阻塞的任务中离开时, 可能需要销毁与之相关的对象并清理有关的资源。正因为这样, 在一个任务的run()中间跳出更像是抛出一个异常, 所以在ZThread库中, 异常被用于此类退出。(这样处于不适当使用异常的边缘, 因为这意味着经常把异常用于控制流。)<sup>①</sup> 为了在以此方式结束一个任务时能返回到一个已知的正确状态, 要谨慎地考虑代码的执行路径, 在catch子句中正确清除所有的东西。在本节, 读者会看到就这些问题的介绍。

为了终止一个阻塞的线程, ZThread库提供了Thread::interrupt()函数。这个函数用来为那类线程设置中断状态 (interrupted status)。一个使用了中断状态设置的线程, 如果已经被阻塞或尝试进行阻塞操作时将会抛出一个InterruptedException异常。当异常被抛出或者假如任务调用了Thread::interrupt()时, 中断状态将重新设置。正如读者所见, Thread::interrupt()提供了不用抛出异常而离开run()函数中循环的第2条途径。

这里的例子显示了interrupt()的基本功能:

```

//: C11:Interrupting.cpp
// Interrupting a blocked thread.
//{L} ZThread
#include <iostream>

```

① 无论如何, 在ZThread中异常绝不会被异步发送。因此退出中间指令或函数调用不会有危险。并且只要我们使用Guard模板获得互斥锁, 那么如果抛出异常的话, 互斥锁会被自动释放。

```

#include "zthread/Thread.h"
using namespace ZThread;
using namespace std;

class Blocked : public Runnable {
public:
    void run() {
        try {
            Thread::sleep(1000);
            cout << "Waiting for get() in run():";
            cin.get();
        } catch(Interrupted_Exception&) {
            cout << "Caught Interrupted_Exception" << endl;
            // Exit the task
        }
    }
};

int main(int argc, char* argv[]) {
    try {
        Thread t(new Blocked);
        if(argc > 1)
            Thread::sleep(1100);
        t.interrupt();
    } catch(Synchronization_Exception& e) {
        cerr << e.what() << endl;
    }
} ///:~

```

可以看到，除了将插入数据到`cout`之外，阻塞还能发生在`run()`函数中包含的其他两个地点：即对`Thread::sleep(1000)`和`cin.get()`的调用。可给程序传递任何命令行参数，可以通知`main()`休眠足够长的时间，以便任务到时候能结束它的`sleep()`和调用`cin.get()`。<sup>①</sup>如果不给程序传递参数，就会跳过`main()`中的`sleep()`。在这里，在任务休眠时发生了对函数`interrupt()`的调用。读者将会看到，这将导致`Interrupted_Exception`异常被抛出。如果给程序一个命令行参数，就会发现如果一个任务被阻塞在IO操作上，它不能被中断。也就是说，除了IO操作，一个任务可以从任何阻塞操作中中断出来。<sup>②</sup>

如果正在创建一个执行IO操作的线程，这还是让人有点困惑，因为这意味着I/O有使多线程处理程序死锁的潜在可能性。问题在于，再次强调，在设计思想上C++没有被设计成使用线程处理；恰恰相反，它假装线程处理并不存在。因此，输入输出流库不是线程友好（thread-friendly）的。如果新的C++标准决定增加对线程的支持，输入输出流库也许需要重新考虑其处理方法。

#### 1. 被一个互斥锁阻塞

如果试图调用一个函数，而该函数的互斥锁已经被别的线程获得了，那么这个调用该函数的任务就会被挂起，直到该互斥锁变成可获得时为止。下面的例子测试了这种阻塞是否可被中断。

```

//: C11:Interrupting2.cpp
// Interrupting a thread blocked
// with a synchronization guard.
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
#include "zthread/Mutex.h"

```

① 实际上，`sleep()`只提供最小的延迟，不是保证延迟，所以可能（尽管不可思议）`sleep(1100)`会在`sleep(1000)`之前被唤醒。

② C++标准中没有说明在IO操作期间中断不能出现。然而大多数实现不支持它。

```

#include "zthread/Guard.h"
using namespace ZThread;
using namespace std;

class BlockedMutex {
    Mutex lock;
public:
    BlockedMutex() {
        lock.acquire();
    }
    void f() {
        Guard<Mutex> g(lock);
        // This will never be available
    }
};

class Blocked2 : public Runnable {
    BlockedMutex blocked;
public:
    void run() {
        try {
            cout << "Waiting for f() in BlockedMutex" << endl;
            blocked.f();
        } catch(Interrupted_Exception& e) {
            cerr << e.what() << endl;
            // Exit the task
        }
    }
};

int main(int argc, char* argv[]) {
    try {
        Thread t(new Blocked2);
        t.interrupt();
    } catch(Synchronization_Exception& e) {
        cerr << e.what() << endl;
    }
} ///:~

```

**BlockedMutex**类有一个构造函数，它获得对象自己的互斥锁**Mutex**并且绝不释放它。由于这个原因，如果试图调用**f()**，总会被阻塞，因为该互斥锁**Mutex**不能被获得。在**Blocked2**中，**run()**函数将因此停止在对**blocked.f()**的调用上。当运行程序时就会看到，和IO流的调用不同，**interrupt()**能够跳出已被一个互斥锁阻塞的调用。<sup>⑨</sup>

## 2. 中断检查

注意，当在一个线程上调用**interrupt()**时，中断仅发生在任务进入一个阻塞操作的那一刻，或者已经在一个阻塞操作内（正如你所知道的，假如在IO的情况下，就会陷在里面）。但是，编写什么样的代码，才能使是否产生这样的阻塞调用依赖于它的运行条件呢？如果只能通过在一个被阻塞的调用上抛出异常来退出，也许始终不能离开**run()**循环。因此，假如调用**interrupt()**来停止一个任务，如果在**run()**循环没有发生任何阻塞调用，该任务就需要另外的机会来退出。

中断状态（interrupted status）提供了这样的机会，它通过调用**interrupt()**进行设置。而调用**interrupted()**来检查中断状态，这不仅能告知**interrupt()**是否已经被调用，它也会清除中断状态。清除中断状态可以确保整个架构不会两次通知正被中断的任务。它会用一个

⑨ 注意，尽管不太可能，对**t.interrupt()**的调用实际可以发生在对**blocked.f()**的调用之前。

**Interrupted\_Exception**异常或者一个成功的**Thread::interrupted()**测试来通知使用者。如果想再次检查是否被中断了,在调用**Thread::interrupted()**时可以把测试结果存储起来。

下面的例子显示了当设置了中断状态时,**run()**函数中在处理阻塞和非阻塞两种可能性的情况下所要使用的典型的习语:

```

//: C11:Interrupting3.cpp {RunByHand}
// General idiom for interrupting a task.
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
using namespace ZThread;
using namespace std;

const double PI = 3.14159265358979323846;
const double E = 2.7182818284590452354;

class NeedsCleanup {
    int id;
public:
    NeedsCleanup(int ident) : id(ident) {
        cout << "NeedsCleanup " << id << endl;
    }
    ~NeedsCleanup() {
        cout << "~NeedsCleanup " << id << endl;
    }
};

class Blocked3 : public Runnable {
    volatile double d;
public:
    Blocked3() : d(0.0) {}
    void run() {
        try {
            while(!Thread::interrupted()) {
                point1:
                NeedsCleanup n1(1);
                cout << "Sleeping" << endl;
                Thread::sleep(1000);
                point2:
                NeedsCleanup n2(2);
                cout << "Calculating" << endl;
                // A time-consuming, non-blocking operation:
                for(int i = 1; i < 100000; i++)
                    d = d + (PI + E) / (double)i;
            }
            cout << "Exiting via while() test" << endl;
        } catch(Interrupted_Exception&) {
            cout << "Exiting via Interrupted_Exception" << endl;
        }
    }
};

int main(int argc, char* argv[]) {
    if(argc != 2) {
        cerr << "usage: " << argv[0]
            << " delay-in-milliseconds" << endl;
        exit(1);
    }
    int delay = atoi(argv[1]);
    try {
        Thread t(new Blocked3);
        Thread::sleep(delay);
    }
}

```

```

        t.interrupt();
    } catch(Synchronization_Exception& e) {
        cerr << e.what() << endl;
    }
} ///:~

```

如果采用抛出异常来离开循环，**NeedsCleanup**类强调了对相关资源进行正确清理的必要性。注意，在**Blocked3::run()**中没有定义指针，那是为了异常的安全，所有的资源必须封装在基于栈的对象中，以便异常处理器可以调用析构函数来自动清理它们。

必须在调用**interrupt()**之前给程序传递一个命令行参数，此参数为用毫秒表示的延迟时间。使用不同的延迟，能从循环中不同的地点退出**Blocked3::run()**函数：从正处于阻塞状态的**sleep()**调用中退出，以及从非阻塞状态的数学计算中退出。可以看到，如果**interrupt()**在标签**point2**后被调用（非阻塞操作期间）。首先循环已经完成，其次所有的本地对象被析构，最后循环经由**while**语句在顶部退出。然而，如果**interrupt()**在**point1**和**point2**之间被调用（在**while**语句之后，但是在阻塞操作**sleep()**之前或之中），任务通过**Interrupted\_Exception**异常退出。在这种情况下，只有在异常被抛出的位置之前已经创建完成的栈对象才会被清理，并且有机会在**catch**子句中完成其他的清理操作。

设计用来响应**interrupt()**函数的类必须建立一种策略，以便保证它能保持一致的状态。这通常意味着，所有的资源获取都要封装在基于栈的对象中，以便无论**run()**循环如何退出，对象的析构函数都会被调用。如果正确地做了，像这样的代码一定是优雅的。在没有向对象接口中加入任何特别的函数的情况下，可以创建出完全封装了其同步机制，但仍能对外部激励（通过**interrupt()**）有响应的组件。

## 11.7 线程间协作

正如读者所看到的，当使用线程在同一时刻运行多个任务时，可以使用互斥锁来同步两个任务的行为的方法，来阻止一个任务干扰另一个任务的资源。也就是说，如果两个任务对一个共享资源（通常是内存）相互争夺，就要使用互斥锁来保证在同一时刻只允许一个任务访问那个资源。

在这个问题解决之后，可以继续考虑线程间协作的问题，以便多个线程能一起工作来共同解决某个问题。现在问题不在于线程之间的彼此干扰，而在于其和谐工作，由于问题的某一部分必须在另外一部分能被解决之前解决完毕。这更像是一个工程进度表：必须先挖房屋的地基，但是钢结构构件的铺设和混凝土构件可以并行建造，这些任务都必须在混凝土基础浇注之前完成。管道设备必须在混凝土平板浇注好之前放置好，而混凝土平板要在开始搭建框架结构之前安置，等等。这些任务中有些可以并行进行，但是某些步骤则要求在完成其他所有任务之后才能继续进行。

这些任务协作时的关键问题是这些任务间的“握手”。为完成这个握手过程，使用相同的基础：互斥机制，互斥机制在这种情况下可以保证只有一个任务响应信号。这就消除了任何可能的竞争条件。要熟练掌握互斥锁，这里为任务增加了一个方法，让它把自己挂起来，直到某些外部状态发生改变（例如“管道设备现在已经就位”），表明此时任务可以向前进行。在本节中，读者会看到任务间的握手问题，在握手期间会出现的问题，以及这些问题相应的解决方法。

### 11.7.1 等待和信号

在Zthread库中，使用互斥锁并允许任务挂起的基类是**Condition**，可以通过在条件**Condition**上调用**wait()**挂起一个任务。当外部状态发生改变时，这种改变也许意味着某个



任务应该继续进行处理。调用信号函数**signal()**可以通知该任务而唤醒它，或者调用**broadcast()**，而唤醒所有在那个**Condition**对象上被挂起的任务。

**wait()**有两种形式。第1种形式接受一个毫秒数作为参数，这个参数与**sleep()**函数中的参数有相同的含义：“在这段时间暂停”。**wait()**的第2种形式不要参数；这种形式更常见。这两种形式的**wait()**都会释放被**Condition**对象所控制的互斥锁 **Mutex**，并且会挂起线程直到**Condition**对象收到一个**signal()**或者**broadcast()**。如果超时，第1种形式在接收到**signal()**或**broadcast()**之前也可以结束。

因为**wait()**会释放**Mutex**，这意味着该**Mutex**可以被其他线程获得。因此，当调用**wait()**时，就相当于说：“现在已经做完了该做的所有事情，我将在此等待，但是我希望如果其他同步操作可以执行的允许它们执行。”

典型的情况是，当在等待某个条件的改变时就使用**wait()**，而这个条件的改变在当前函数之外的因素控制之下进行。（这些条件常常会被另一个线程改变。）在线程内检测条件时，你不希望进行空循环等待；这就是所谓的“忙等待”，而“忙等待”通常会大量占用CPU周期。因此，**wait()**在等待外部条件变化时挂起线程，只在**signal()**或**broadcast()**被调用时（暗示某些相关事件已经发生），唤醒线程并检测发生的变化。因此，**wait()**为同步线程之间的活动提供了一种方法。

下面看一个简单的例子。**WaxOMatic.cpp**有两个进程：一个进程给**Car**上蜡，另一个进程给**Car**抛光。抛光进程在上蜡进程完成前不能进行其工作，并且上蜡进程在汽车可以再穿上另一个蜡外套之前必须等待直到抛光进程完成。**WaxOn**和**WaxOff**都使用了**Car**对象，这个**Car**对象包含了一个用于挂起一个在**waitForWaxing()**或**waitForBuffing()**内的线程的**Condition**。

```
//: C11:WaxOMatic.cpp {RunByHand}
// Basic thread cooperation.
//{L} ZThread
#include <iostream>
#include <string>
#include "zthread/Thread.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"
#include "zthread/Condition.h"
#include "zthread/ThreadedExecutor.h"
using namespace ZThread;
using namespace std;

class Car {
    Mutex lock;
    Condition condition;
    bool waxOn;
public:
    Car() : condition(lock), waxOn(false) {}
    void waxed() {
        Guard<Mutex> g(lock);
        waxOn = true; // Ready to buff
        condition.signal();
    }
    void buffed() {
        Guard<Mutex> g(lock);
        waxOn = false; // Ready for another coat of wax
        condition.signal();
    }
    void waitForWaxing() {
```





```

        Guard<Mutex> g(lock);
        while(waxOn == false)
            condition.wait();
    }
    void waitForBuffing() {
        Guard<Mutex> g(lock);
        while(waxOn == true)
            condition.wait();
    }
};

class WaxOn : public Runnable {
    CountedPtr<Car> car;
public:
    WaxOn(CountedPtr<Car>& c) : car(c) {}
    void run() {
        try {
            while(!Thread::interrupted()) {
                cout << "Wax On!" << endl;
                Thread::sleep(200);
                car->waxed();
                car->waitForBuffing();
            }
        } catch(Interrupted_Exception&) { /* Exit */ }
        cout << "Ending Wax On process" << endl;
    }
};

class WaxOff : public Runnable {
    CountedPtr<Car> car;
public:
    WaxOff(CountedPtr<Car>& c) : car(c) {}
    void run() {
        try {
            while(!Thread::interrupted()) {
                car->waitForWaxing();
                cout << "Wax Off!" << endl;
                Thread::sleep(200);
                car->buffed();
            }
        } catch(Interrupted_Exception&) { /* Exit */ }
        cout << "Ending Wax Off process" << endl;
    }
};

int main() {
    cout << "Press <Enter> to quit" << endl;
    try {
        CountedPtr<Car> car(new Car);
        ThreadedExecutor executor;
        executor.execute(new WaxOff(car));
        executor.execute(new WaxOn(car));
        cin.get();
        executor.interrupt();
    } catch(Synchronization_Exception& e) {
        cerr << e.what() << endl;
    }
} ///:~

```

在**Car**的构造函数中，一个互斥锁**Mutex**被封装于**Condition**对象中，以便**Mutex**可以用于管理任务间的通信。然而，**Condition**对象不包含有关进程状态的信息，所以还要管理另外的信息用来指出进程的状态。在这里，**Car**有一个**bool waxOn**，这个布尔变量指出上蜡、

抛光进程的状态。

在**waitForWaxing()**中检查**waxOn**标志, 如果它是**false**则调用中的线程通过调用**Condition**对象上的**wait()**被挂起。重要的是, 这发生在一个被保护的子句中, 在这个子句中该线程获得了互斥锁 (在这里, 是通过创建一个**Guard**对象获得的)。当调用**wait()**时, 该线程被挂起并释放互斥锁。释放互斥锁是必要的, 因为为了安全地改变对象的状态 (比如, 把**waxOn**的值变为**true**, 为了使被挂起的线程继续进行下去, 这是必须做的), 互斥锁必须能够被一些其他任务获得。在本例中, 当其他线程调用**waxed()**来告知被挂起的线程该去做某个工作的时候, 互斥锁必须能获得以便把**waxOn**的值变为**true**。然后, **waxed()**向**Condition**对象发送一个信号**signal()**, 由它来唤醒那个在调用**wait()**中被挂起的线程。虽然可以在一个被保护的子句中调用信号**signal()**——就像这里一样——但并不要求这样做。<sup>①</sup>

为了从等待**wait()**中唤醒一个线程, 必须首先重新获得其在进入**wait()**时释放的互斥锁。直到互斥锁变成可用之前, 该线程不会被唤醒。

**wait()**的调用被置于一个**while**循环内部, 用这个循环来检查相关的条件。这很重要, 基于以下两个原因:<sup>②</sup>

- 很可能当某个线程得到一个信号**signal()**时, 其他一些条件可能已经改变了, 但这些条件在这里与调用**wait()**的原因无关。如果有这种情况, 该线程在其相关的条件改变之前将再一次被挂起。
- 在该线程从其**wait()**函数中醒来之时, 可能另外某个任务改变了一些条件, 因此这个线程就不能或者没兴趣在此时执行其操作了。再次强调, 它应再次调用**wait()**而被重新挂起。

因为这两个原因在调用**wait()**时总会出现, 故总要编写在**while**循环内调用**wait()**的一段程序来测试与线程相关的条件。

**WaxOn::run()**代表给汽车上蜡进程中的第1步, 所以它执行其操作 (调用**sleep()**来模拟上蜡所需要的时间)。然后它告知汽车上蜡完毕, 并调用**waitForBuffing()**, 该函数使用**wait()**挂起线程, 直到**WaxOff**进程为汽车调用**buffed()**, 改变状态并调用**notify()**。另一方面, **WaxOff::run()**立即迁移到**waitForWaxing()**, 并因此被挂起直到由**WaxOn**将上蜡工作完成并且**waxed()**被调用。当运行此程序时可以看到, 将控制权在两个线程之间来回传递, 从而使这两步进程交替重复执行。当按下回车 (<Enter>) 键时, **interrupt()**停止这两个线程的运行——当为一个执行器对象**Executor**调用**interrupt()**时, 它为其控制下的所有线程调用**interrupt()**。

### 11.7.2 生产者-消费者关系

线程处理问题中的一个常见的情形是生产者-消费者 (producer-consumer) 关系, 一个任务创建对象而另一个任务消费这些对象。在这种情况下, 要确定 (在其他事件中) 进行消费的任务不会意外遗漏掉已产生的任何对象。

为了说明该问题, 考虑一个有3个任务的机器: 一个任务是制作烤面包, 一个任务是给烤

① 这与Java相反, 在Java中必须持有锁才能调用**notify()** (Java版的**signal()**)。尽管Posix线程不要求必须持有锁才能调用**signal()**或**broadcast()**, 但是这种做法是推荐的做法。ZThread库松散基于Posix线程。

② 在一些平台上有第3种办法跳出**wait()**, 即所谓伪唤醒 (spurious wakeup)。一个伪唤醒本质上意味着一个线程过早地停止了阻塞 (当等待一个条件变量或信号量时) 而没有被**signal()**或**broadcast()**激活。线程就像是自己醒过来一样。伪唤醒存在的原因是, 在某些平台上实现POSIX线程或类似的东西, 并不像它在某些平台上那样直截了当。对这些平台来说, 允许伪唤醒能够简化建立类似pthread库的工作。ZThread中不存在伪唤醒, 因为该库弥补并对用户隐藏了这些问题。

面包抹黄油，还有一个任务是往抹好黄油的烤面包上抹果酱。

```

//: C11:ToastOMatic.cpp {RunByHand}
// Problems with thread cooperation.
//{L} ZThread
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "zthread/Thread.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"
#include "zthread/Condition.h"
#include "zthread/ThreadedExecutor.h"
using namespace ZThread;
using namespace std;

// Apply jam to buttered toast:
class Jammer : public Runnable {
    Mutex lock;
    Condition butteredToastReady;
    bool gotButteredToast;
    int jammed;
public:
    Jammer() : butteredToastReady(lock) {
        gotButteredToast = false;
        jammed = 0;
    }
    void moreButteredToastReady() {
        Guard<Mutex> g(lock);
        gotButteredToast = true;
        butteredToastReady.signal();
    }
    void run() {
        try {
            while(!Thread::interrupted()) {
                {
                    Guard<Mutex> g(lock);
                    while(!gotButteredToast)
                        butteredToastReady.wait();
                    ++jammed;
                }
                cout << "Putting jam on toast " << jammed << endl;
                {
                    Guard<Mutex> g(lock);
                    gotButteredToast = false;
                }
            }
        } catch(Interrupted_Exception&) { /* Exit */ }
        cout << "Jammer off" << endl;
    }
};

// Apply butter to toast:
class Butterer : public Runnable {
    Mutex lock;
    Condition toastReady;
    CountedPtr<Jammer> jammer;
    bool gotToast;
    int buttered;
public:
    Butterer(CountedPtr<Jammer>& j)
        : toastReady(lock), jammer(j) {
        gotToast = false;
    }
};

```



```

    buttered = 0;
}
void moreToastReady() {
    Guard<Mutex> g(lock);
    gotToast = true;
    toastReady.signal();
}
void run() {
    try {
        while(!Thread::interrupted()) {
            {
                Guard<Mutex> g(lock);
                while(!gotToast)
                    toastReady.wait();
                ++buttered;
            }
            cout << "Buttering toast " << buttered << endl;
            jammer->moreButteredToastReady();
            {
                Guard<Mutex> g(lock);
                gotToast = false;
            }
        }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Butterer off" << endl;
}
};

class Toaster : public Runnable {
    CountedPtr<Butterer> butterer;
    int toasted;
public:
    Toaster(CountedPtr<Butterer>& b) : butterer(b) {
        toasted = 0;
    }
    void run() {
        try {
            while(!Thread::interrupted()) {
                Thread::sleep(rand()/(RAND_MAX/5)*100);
                // ...
                // Create new toast
                // ...
                cout << "New toast " << ++toasted << endl;
                butterer->moreToastReady();
            }
        } catch(Interrupted_Exception&) { /* Exit */ }
        cout << "Toaster off" << endl;
    }
};

int main() {
    srand(time(0)); // Seed the random number generator
    try {
        cout << "Press <Return> to quit" << endl;
        CountedPtr<Jammer> jammer(new Jammer);
        CountedPtr<Butterer> butterer(new Butterer(jammer));
        ThreadedExecutor executor;
        executor.execute(new Toaster(butterer));
        executor.execute(butterer);
        executor.execute(jammer);
        cin.get();
        executor.interrupt();
    }
}

```



```

    } catch(Synchronization_Exception& e) {
        cerr << e.what() << endl;
    }
} ///:~

```

这些类以逆序定义，这样做的目的是简化前向引用（forward-referencing）的操作问题。

**Jammer**和**Butterer**都包含一个**Mutex**对象、一个**Condition**对象和一些内部状态信息。通过改变这些内部状态信息的状态，来指出进程要被挂起或恢复执行。（**Toaster**不需要这些，因为它是生产者，无需等待任何事情。）两个**run()**函数都执行同一个操作，设置一个状态标志，然后调用**wait()**来挂起任务。**moreToastReady()**和**moreButteredToastReady()**函数改变各自的状态标志，以指示某些事情已经发生了改变，进程现在要考虑恢复执行，然后调用信号**signal()**唤醒该线程。

本例与前面例子的不同之处在于，至少从概念上讲，这里生产了一些东西：烤面包。烤面包的生产率有点随机化，这就增加了不确定性。读者将会看到，在运行程序时可能会出错，因为会有许多片烤面包掉在地板上——没抹黄油，也没抹果酱。

### 11.7.3 用队列解决线程处理的问题

线程处理问题常常基于需要对任务进行串行化上——也就是说，要使事情有序地进行处理。**ToastOMatic.cpp**不仅必要注意让事情有序，还必须能够加工好烤面包片，而且在此期间不用担心它会掉到地板上。使用队列可以采用同步的方式访问其内部元素，这样就可以解决很多线程处理问题：

```

//: C11:TQueue.h
#ifndef TQUEUE_H
#define TQUEUE_H
#include <deque>
#include "zthread/Thread.h"
#include "zthread/Condition.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"

template<class T> class TQueue {
    ZThread::Mutex lock;
    ZThread::Condition cond;
    std::deque<T> data;
public:
    TQueue() : cond(lock) {}
    void put(T item) {
        ZThread::Guard<ZThread::Mutex> g(lock);
        data.push_back(item);
        cond.signal();
    }
    T get() {
        ZThread::Guard<ZThread::Mutex> g(lock);
        while(data.empty())
            cond.wait();
        T returnVal = data.front();
        data.pop_front();
        return returnVal;
    }
};
#endif // TQUEUE_H ///:~

```

这是通过在标准C++库的双端队列**deque**上添加下面的内容建立起来的：

1) 加入同步以确保在同一时刻不会有两个线程添加对象。



2) 加入**wait( )**和**signal( )**以便在队列为空时让消费者线程自动挂起，并在有多个元素可用时恢复执行。

这些相对量较少的代码能解决为数可观的问题。<sup>⊖</sup>

这里有个简单的测试程序，将对**LiftOff**对象的串行化执行进行测试。消费者是**LiftOffRunner**，它把每个**LiftOff**对象从**TQueue**中取出来并直接执行。（也就是说，它通过显式调用**run( )**来使用自己的线程，而不是为每个任务启动一个新线程。）

```
//: C11:TestTQueue.cpp {RunByHand}
//{L} ZThread
#include <string>
#include <iostream>
#include "TQueue.h"
#include "zthread/Thread.h"
#include "LiftOff.h"
using namespace ZThread;
using namespace std;

class LiftOffRunner : public Runnable {
    TQueue<LiftOff*> rockets;
public:
    void add(LiftOff* lo) { rockets.put(lo); }
    void run() {
        try {
            while(!Thread::interrupted()) {
                LiftOff* rocket = rockets.get();
                rocket->run();
            }
        } catch(Interrupted_Exception&) { /* Exit */ }
        cout << "Exiting LiftOffRunner" << endl;
    }
};

int main() {
    try {
        LiftOffRunner* lor = new LiftOffRunner;
        Thread t(lor);
        for(int i = 0; i < 5; i++)
            lor->add(new LiftOff(10, i));
        cin.get();
        lor->add(new LiftOff(10, 99));
        cin.get();
        t.interrupt();
    } catch(Synchronization_Exception& e) {
        cerr << e.what() << endl;
    }
} ///:~
```

任务被**main( )**函数置于**TQueue**队列上，被**LiftOffRunner**从**TQueue**队列上取走。注意，**LiftOffRunner**可以忽略同步问题，因为这些问题由**TQueue**来解决。

适当地进行烘烤

为解决**ToastOMatic.cpp**中存在的问题，我们可以在加工进程期间使用**TQueue**管理烤面包。为了做到这点，需要实际的烤面包对象，它们保持并显示了其状态：

⊖ 注意，如果读者由于某些原因停止了读，写者将继续写入直到系统内存用完。如果这是用户的程序存在的一个问题，用户可以添加所允许的最大元素计数，队列满时写者线程将被阻塞。

```

//: C11:ToastOMaticMarkII.cpp {RunByHand}
// Solving the problems using TQueues.
//{L} ZThread
#include <iostream>
#include <string>
#include <cstdlib>
#include <ctime>
#include "zthread/Thread.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"
#include "zthread/Condition.h"
#include "zthread/ThreadedExecutor.h"
#include "TQueue.h"
using namespace ZThread;
using namespace std;

class Toast {
    enum Status { DRY, BUTTERED, JAMMED };
    Status status;
    int id;
public:
    Toast(int idn) : status(DRY), id(idn) {}
#ifdef __DMC__ // Incorrectly requires default
    Toast() { assert(0); } // Should never be called
#endif
    void butter() { status = BUTTERED; }
    void jam() { status = JAMMED; }
    string getStatus() const {
        switch(status) {
            case DRY: return "dry";
            case BUTTERED: return "buttered";
            case JAMMED: return "jammed";
            default: return "error";
        }
    }
    int getId() { return id; }
    friend ostream& operator<<(ostream& os, const Toast& t) {
        return os << "Toast " << t.id << ": " << t.getStatus();
    }
};

typedef CountedPtr< TQueue<Toast> > ToastQueue;

class Toaster : public Runnable {
    ToastQueue toastQueue;
    int count;
public:
    Toaster(ToastQueue& tq) : toastQueue(tq), count(0) {}
    void run() {
        try {
            while(!Thread::interrupted()) {
                int delay = rand()/(RAND_MAX/5)*100;
                Thread::sleep(delay);
                // Make toast
                Toast t(count++);
                cout << t << endl;
                // Insert into queue
                toastQueue->put(t);
            }
        } catch(Interrupted_Exception&) { /* Exit */ }
        cout << "Toaster off" << endl;
    }
};

```



```

// Apply butter to toast:
class Butterer : public Runnable {
    ToastQueue dryQueue, butteredQueue;
public:
    Butterer(ToastQueue& dry, ToastQueue& buttered)
        : dryQueue(dry), butteredQueue(buttered) {}
    void run() {
        try {
            while(!Thread::interrupted()) {
                // Blocks until next piece of toast is available:
                Toast t = dryQueue->get();
                t.butter();
                cout << t << endl;
                butteredQueue->put(t);
            }
        } catch(Interrupted_Exception&) { /* Exit */ }
        cout << "Butterer off" << endl;
    }
};

// Apply jam to buttered toast:
class Jammer : public Runnable {
    ToastQueue butteredQueue, finishedQueue;
public:
    Jammer(ToastQueue& buttered, ToastQueue& finished)
        : butteredQueue(buttered), finishedQueue(finished) {}
    void run() {
        try {
            while(!Thread::interrupted()) {
                // Blocks until next piece of toast is available:
                Toast t = butteredQueue->get();
                t.jam();
                cout << t << endl;
                finishedQueue->put(t);
            }
        } catch(Interrupted_Exception&) { /* Exit */ }
        cout << "Jammer off" << endl;
    }
};

// Consume the toast:
class Eater : public Runnable {
    ToastQueue finishedQueue;
    int counter;
public:
    Eater(ToastQueue& finished)
        : finishedQueue(finished), counter(0) {}
    void run() {
        try {
            while(!Thread::interrupted()) {
                // Blocks until next piece of toast is available:
                Toast t = finishedQueue->get();
                // Verify that the toast is coming in order,
                // and that all pieces are getting jammed:
                if(t.getId() != counter++ ||
                   t.getStatus() != "jammed") {
                    cout << ">>>> Error: " << t << endl;
                    exit(1);
                } else
                    cout << "Chomp! " << t << endl;
            }
        } catch(Interrupted_Exception&) { /* Exit */ }
        cout << "Eater off" << endl;
    }
};

```



```

    }
};

int main() {
    srand(time(0)); // Seed the random number generator
    try {
        ToastQueue dryQueue(new TQueue<Toast>),
            butteredQueue(new TQueue<Toast>),
            finishedQueue(new TQueue<Toast>);
        cout << "Press <Return> to quit" << endl;
        ThreadedExecutor executor;
        executor.execute(new Toaster(dryQueue));
        executor.execute(new Butterer(dryQueue,butteredQueue));
        executor.execute(
            new Jammer(butteredQueue, finishedQueue));
        executor.execute(new Eater(finishedQueue));
        cin.get();
        executor.interrupt();
    } catch(Synchronization_Exception& e) {
        cerr << e.what() << endl;
    }
} ///:~

```

在这个解决方案中，两件事情会马上变得很明显：第一，在每个**Runnable**类中代码的数量和复杂性通过队列**TQueue**的使用会显著减少，因为进行保护、通信，以及**wait()** / **signal()**操作现在都由**TQueue**来维护。**Runnable**类不再拥有任何**Mutex**或**Condition**对象。第二，类之间的耦合被消除了，因为每个类只与它的**TQueue**通信。注意，现在类的定义次序是独立的。较少的代码和较少的耦合总归是一件好事，这暗示着在这里**TQueue**的使用有积极作用，就像在大多数问题中它所做的那样。

#### 11.7.4 广播

**signal()**函数唤醒了一个正在等待**Condition**对象的线程。然而，也许会有多个线程在等待某个相同的条件对象，在这种情况下需要使用**broadcast()**而不是**signal()**把这些线程都唤醒。

现在考虑一个假想的制造汽车的机器人流水线，作为一个例子它集中体现了本章中的许多概念。每辆**Car**将在几个阶段内装配完成，在本例中将看到这样一个阶段：底盘制造好后，在这段时间里安装附属的发动机、驱动传动装置（drive train）和车轮。通过一个**CarQueue**将**Car**从一个地方传送到另一个地方，**CarQueue**是一个**TQueue**的类型。一个**Director**从进来的**CarQueue**队列中取出每辆**Car**（作为一个未加工的底盘）并放置在一个**Cradle**中，所有工作都在这里完成。在这个地方，主管**Director**通知所有正在等待的机器人（使用广播**broadcast()**），**Car**已经在**Cradle**中准备就绪，机器人们可以进行装配工作了。三种类型的机器人开始进行工作，当它们完成任务时给**Cradle**发送一个消息。**Director**一直等到所有任务都完成后，把**Car**放到出去的**CarQueue**队列上传送到下一个工序。在这里，出去的**CarQueue**队列的消费者是个**Reporter**对象，它仅打印该**Car**，说明那个任务已正确地完成了。

```

//: C11:CarBuilder.cpp {RunByHand}
// How broadcast() works.
//{L} ZThread
#include <iostream>
#include <string>
#include "zthread/Thread.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"
#include "zthread/Condition.h"
#include "zthread/ThreadedExecutor.h"

```

```

#include "TQueue.h"
using namespace ZThread;
using namespace std;

class Car {
    int id;
    bool engine, driveTrain, wheels;
public:
    Car(int idn) : id(idn), engine(false),
        driveTrain(false), wheels(false) {}
    // Empty Car object:
    Car() : id(-1), engine(false),
        driveTrain(false), wheels(false) {}
    // Unsynchronized -- assumes atomic bool operations:
    int getId() { return id; }
    void addEngine() { engine = true; }
    bool engineInstalled() { return engine; }
    void addDriveTrain() { driveTrain = true; }
    bool driveTrainInstalled() { return driveTrain; }
    void addWheels() { wheels = true; }
    bool wheelsInstalled() { return wheels; }
    friend ostream& operator<<(ostream& os, const Car& c) {
        return os << "Car " << c.id << " ["
            << " engine: " << c.engine
            << " driveTrain: " << c.driveTrain
            << " wheels: " << c.wheels << " ]";
    }
};

typedef CountedPtr< TQueue<Car> > CarQueue;

class ChassisBuilder : public Runnable {
    CarQueue carQueue;
    int counter;
public:
    ChassisBuilder(CarQueue& cq) : carQueue(cq), counter(0) {}
    void run() {
        try {
            while(!Thread::interrupted()) {
                Thread::sleep(1000);
                // Make chassis:
                Car c(counter++);
                cout << c << endl;
                // Insert into queue
                carQueue->put(c);
            }
        } catch(Interrupted_Exception&) { /* Exit */ }
        cout << "ChassisBuilder off" << endl;
    }
};

class Cradle {
    Car c; // Holds current car being worked on
    bool occupied;
    Mutex workLock, readyLock;
    Condition workCondition, readyCondition;
    bool engineBotHired, wheelBotHired, driveTrainBotHired;
public:
    Cradle()
        : workCondition(workLock), readyCondition(readyLock) {
        occupied = false;
        engineBotHired = true;
        wheelBotHired = true;
        driveTrainBotHired = true;
    }
};

```

```

    }
    void insertCar(Car chassis) {
        c = chassis;
        occupied = true;
    }
    Car getCar() { // Can only extract car once
        if(!occupied) {
            cerr << "No Car in Cradle for getCar()" << endl;
            return Car(); // "Null" Car object
        }
        occupied = false;
        return c;
    }
    // Access car while in cradle:
    Car* operator->() { return &c; }
    // Allow robots to offer services to this cradle:
    void offerEngineBotServices() {
        Guard<Mutex> g(workLock);
        while(engineBotHired)
            workCondition.wait();
        engineBotHired = true; // Accept the job
    }
    void offerWheelBotServices() {
        Guard<Mutex> g(workLock);
        while(wheelBotHired)
            workCondition.wait();
        wheelBotHired = true; // Accept the job
    }
    void offerDriveTrainBotServices() {
        Guard<Mutex> g(workLock);
        while(driveTrainBotHired)
            workCondition.wait();
        driveTrainBotHired = true; // Accept the job
    }
    // Tell waiting robots that work is ready:
    void startWork() {
        Guard<Mutex> g(workLock);
        engineBotHired = false;
        wheelBotHired = false;
        driveTrainBotHired = false;
        workCondition.broadcast();
    }
    // Each robot reports when their job is done:
    void taskFinished() {
        Guard<Mutex> g(readyLock);
        readyCondition.signal();
    }
    // Director waits until all jobs are done:
    void waitUntilWorkFinished() {
        Guard<Mutex> g(readyLock);
        while(!(c.engineInstalled() && c.driveTrainInstalled()
            && c.wheelsInstalled()))
            readyCondition.wait();
    }
};

typedef CountedPtr<Cradle> CradlePtr;

class Director : public Runnable {
    CarQueue chassisQueue, finishingQueue;
    CradlePtr cradle;
public:
    Director(CarQueue& cq, CarQueue& fq, CradlePtr cr)

```



```

: chassisQueue(cq), finishingQueue(fq), cradle(cr) {}
void run() {
    try {
        while(!Thread::interrupted()) {
            // Blocks until chassis is available:
            cradle->insertCar(chassisQueue->get());
            // Notify robots car is ready for work
            cradle->startWork();
            // Wait until work completes
            cradle->waitUntilWorkFinished();
            // Put car into queue for further work
            finishingQueue->put(cradle->getCar());
        }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Director off" << endl;
}
};

class EngineRobot : public Runnable {
    CradlePtr cradle;
public:
    EngineRobot(CradlePtr cr) : cradle(cr) {}
    void run() {
        try {
            while(!Thread::interrupted()) {
                // Blocks until job is offered/accepted:
                cradle->offerEngineBotServices();
                cout << "Installing engine" << endl;
                (*cradle)->addEngine();
                cradle->taskFinished();
            }
        } catch(Interrupted_Exception&) { /* Exit */ }
        cout << "EngineRobot off" << endl;
    }
};

class DriveTrainRobot : public Runnable {
    CradlePtr cradle;
public:
    DriveTrainRobot(CradlePtr cr) : cradle(cr) {}
    void run() {
        try {
            while(!Thread::interrupted()) {
                // Blocks until job is offered/accepted:
                cradle->offerDriveTrainBotServices();
                cout << "Installing DriveTrain" << endl;
                (*cradle)->addDriveTrain();
                cradle->taskFinished();
            }
        } catch(Interrupted_Exception&) { /* Exit */ }
        cout << "DriveTrainRobot off" << endl;
    }
};

class WheelRobot : public Runnable {
    CradlePtr cradle;
public:
    WheelRobot(CradlePtr cr) : cradle(cr) {}
    void run() {
        try {
            while(!Thread::interrupted()) {
                // Blocks until job is offered/accepted:
                cradle->offerWheelBotServices();
            }
        }
    }
};

```



```

        cout << "Installing Wheels" << endl;
        (*cradle)->addWheels();
        cradle->taskFinished();
    }
} catch(Interrupted_Exception&) { /* Exit */ }
cout << "WheelRobot off" << endl;
}
};

class Reporter : public Runnable {
    CarQueue carQueue;
public:
    Reporter(CarQueue& cq) : carQueue(cq) {}
    void run() {
        try {
            while(!Thread::interrupted()) {
                cout << carQueue->get() << endl;
            }
        } catch(Interrupted_Exception&) { /* Exit */ }
        cout << "Reporter off" << endl;
    }
};

int main() {
    cout << "Press <Enter> to quit" << endl;
    try {
        CarQueue chassisQueue(new TQueue<Car>),
            finishingQueue(new TQueue<Car>);
        CradlePtr cradle(new Cradle);
        ThreadedExecutor assemblyLine;
        assemblyLine.execute(new EngineRobot(cradle));
        assemblyLine.execute(new DriveTrainRobot(cradle));
        assemblyLine.execute(new WheelRobot(cradle));
        assemblyLine.execute(
            new Director(chassisQueue, finishingQueue, cradle));
        assemblyLine.execute(new Reporter(finishingQueue));
        // Start everything running by producing chassis:
        assemblyLine.execute(new ChassisBuilder(chassisQueue));
        cin.get();
        assemblyLine.interrupt();
    } catch(Synchronization_Exception& e) {
        cerr << e.what() << endl;
    }
} ///:~

```

注意，**Car**走了一个捷径：它假设布尔操作是原子的，就像以前讨论过的那样，有时候这是一个安全的假定，但需要周密考虑。<sup>①</sup>每个**Car**从一个未加工的底盘开始，不同的机器人给它装配上不同的部分，当它们去做这件事时要调用适当的“add”函数。

**ChassisBuilder**只是每秒钟创建一个新的**Car**，把它放进**chassisQueue**队列中。**Director**通过把下一个**Car**从**chassisQueue**队列中取出，把它放入**Cradle**，而通知所有机器人去**startWork()**，并通过调用**waitUntilWorkFinished()**挂起自己等一系列操作来管理装配进程。当工作完成时，**Director**把**Car**从**Cradle**中取出并放入**finishingQueue**。

**Cradle**是发送信号操作的关键。互斥锁**Mutex**和**Condition**条件对象控制着两件事情：机器人进行的工作和显示所有的操作是否已经完成。一个特定类型的机器人能够通过调用与其类型相适应的“提供”函数将其服务提供给**Cradle**。在这个地方，机器人线程被挂起，直到

① 详细说明，请参考本章先前关于多处理器和可见性的注释。

**Director**调用开始工作函数**startWork()**，它改变雇佣标志(hiring flag)并调用**broadcast()**来通知所有机器人出来工作。虽然这个系统允许任意数量的机器人提供服务，但每个机器人为做这些工作需要挂起自己的线程。可以想象一个更复杂的系统，在该系统中各种机器人在不同的**Cradle**里面注册自己，并没有被注册进程挂起。然后将它们存在一个对象池中，等待第1个需要完成某种任务的**Cradle**。

每个机器人完成了它的任务(改变进程中**Car**的状态)后，它就调用**taskFinished()**，此函数向**readyCondition**发送一个信号**signal()**，而这正是**Director**在**waitUntilWorkFinished()**函数中所等待的。每次主管(director)线程醒来，都会检查**Car**的状态。如果它仍然未完成，这个线程会被再次挂起。

当**Director**将一个**Car**插入到**Cradle**中时，可以通过运算符**operator->()**在**Car**上执行操作。为了防止多次提取同一辆汽车，用一个标志引发产生一个错误报告。(在**ZThread**库中异常不能跨线程传播。)

在**main()**函数中，随着**ChassisBuilder**开始持续启动进程，所有必需的对象都被创建并且所有的任务都被初始化。(然而，因为**TQueue**的行为，如果它先启动也没关系。)注意，这个程序遵循了本章给出的所有关于对象和任务生存周期的指南，故停止进程是安全的。

## 11.8 死锁

因为线程可以变为阻塞，且因为对象可以拥有互斥锁，这些锁能够阻止线程在锁被释放之前访问这个对象。所以就有可能出现这种情况，某个线程在等待另一个线程而第2个线程又在等待别的线程，以此类推，直到这个链上的最后一个线程回过头来等待第1个线程。这样就会得到一个由互相等待的线程构成的连续的循环，而使任何线程都不能运行。这称为死锁(deadlock)。

如果试图运行一个程序，它立刻就死锁了，人们马上就会知道程序出了问题，并且可以跟踪程序的执行过程来找到问题所在。真正的问题在于，这个程序似乎运行良好，但却隐藏着死锁的可能性。在这种情况下，死锁可能发生但事先却得没有任何征兆，所以它潜伏在程序里，直到客户发现它出其不意地发生了。(并且这个死锁的过程可能很难重复显现。)因此，仔细设计程序来预防死锁是开发并发程序的一个关键的要素。

现在看一个由Edsger Dijkstra虚构的有关死锁的经典问题：哲学家聚餐(dining philosopher)问题。该问题的基本描述指定了5个哲学家(不过这里的例子允许任意数目的哲学家)。这些哲学家将花费他们的部分时间用来进行思考，花费其余部分时间进餐。当他们进行思考时，不需要任何共享资源，但是在他们进餐时使用有限数量的餐具。在原始的问题描述中，餐具就是叉子，每个人需要用两个叉子从桌子中央的碗里取意大利面条，但似乎将餐具说成是筷子更合理。显然，只要每个哲学家进餐就需要两根筷子。

该问题引入了一个难点：作为哲学家，他们只有很少的钱，所以只买得起5根筷子。哲学家们围坐在桌子周围，哲学家与哲学家之间放一根筷子。当一个哲学家想进餐时，他必须同时得到他左边的那根筷子和右边的那根筷子。如果该哲学家的旁边(左边或右边)有人正在使用他所需要的筷子，则我们的这个哲学家就必须等待，直到所需要的筷子变成可用的。

```
//: C11:DiningPhilosophers.h
// Classes for Dining Philosophers.
#ifdef DININGPHILOSOPHERS_H
#define DININGPHILOSOPHERS_H
#include <string>
#include <iostream>
#include <cstdlib>
```

```

#include "zthread/Condition.h"
#include "zthread/Guard.h"
#include "zthread/Mutex.h"
#include "zthread/Thread.h"
#include "Display.h"

class Chopstick {
    ZThread::Mutex lock;
    ZThread::Condition notTaken;
    bool taken;
public:
    Chopstick() : notTaken(lock), taken(false) {}
    void take() {
        ZThread::Guard<ZThread::Mutex> g(lock);
        while(taken)
            notTaken.wait();
        taken = true;
    }
    void drop() {
        ZThread::Guard<ZThread::Mutex> g(lock);
        taken = false;
        notTaken.signal();
    }
};

class Philosopher : public ZThread::Runnable {
    Chopstick& left;
    Chopstick& right;
    int id;
    int ponderFactor;
    ZThread::CountedPtr<Display> display;
    int randSleepTime() {
        if(ponderFactor == 0) return 0;
        return rand()/(RAND_MAX/ponderFactor) * 250;
    }
    void output(std::string s) {
        std::ostringstream os;
        os << *this << " " << s << std::endl;
        display->output(os);
    }
public:
    Philosopher(Chopstick& l, Chopstick& r,
        ZThread::CountedPtr<Display>& disp, int ident, int ponder)
        : left(l), right(r), id(ident), ponderFactor(ponder),
        display(disp) {}
    virtual void run() {
        try {
            while(!ZThread::Thread::interrupted()) {
                output("thinking");
                ZThread::Thread::sleep(randSleepTime());
                // Hungry
                output("grabbing right");
                right.take();
                output("grabbing left");
                left.take();
                output("eating");
                ZThread::Thread::sleep(randSleepTime());
                right.drop();
                left.drop();
            }
        } catch(ZThread::Synchronization_Exception& e) {
            output(e.what());
        }
    }
};

```



```

    }
    friend std::ostream&
    operator<<(std::ostream& os, const Philosopher& p) {
        return os << "Philosopher " << p.id;
    }
};
#endif // DININGPHILOSOPHERS_H ///:~

```

两个哲学家**Philosopher**不可以同时用**take()**拿同一根筷子**Chopstick**, 因为**take()**用一个互斥锁**Mutex**进行同步。另外, 如果筷子已经被一个**Philosopher**占用, 另一个**Philosopher**可以在可用条件**available Condition**上用**wait()**等待, 直到**Chopstick**的当前持有者调用**drop()**放下筷子(这也必须同步以防竞争条件, 并且保证多处理器系统中的内存可见性)使**Chopstick**变为可用的。

每个**Philosopher**持有其左边和右边**Chopstick**对象的引用, 所以可以尝试拿起它们。**Philosopher**的目的是用部分时间进行思考, 用另一部分时间进餐, 在**main()**函数中就是这样表达的。然而读者会注意到, 如果**Philosopher**花很少的时间进行思考, 当他们试图进餐时都来对**Chopstick**进行竞争, 死锁就会更快地发生。所以, 可以这样试验一下, 思考算子**ponderFactor**用于衡量一个**Philosopher**花费在思考和进餐上的时间长度趋势。一个较小的**ponderFactor**将增加死锁的可能性。

在**Philosopher::run()**中, 每个**Philosopher**仅仅不断地重复进行思考和进餐。可以看到**Philosopher**用于思考的时间量是随机的, 然后试图用**take()**拿右边的**Chopstick**, 再用**take()**拿左边的**Chopstick**, 用于进餐的时间量亦是随机的, 然后再次重复这样做。对输出到控制台的操作进行同步, 就像在本章中较早时见到的一样。

这个问题很有趣, 因为它显示了一个程序可能表面上看起来运行正确, 但事实上有死锁的倾向。为了演示这一点, 可以使用命令行参数调整因子来影响进餐哲学家的总数及每个哲学家花费思考的时间。如果有许多哲学家或者他们花费很多时间进行思考, 那么, 虽然有死锁的可能性, 但也许永远也看不到死锁。值为0的命令行参数趋向于使死锁尽快发生。<sup>⑨</sup>

```

//: C11:DeadlockingDiningPhilosophers.cpp {RunByHand}
// Dining Philosophers with Deadlock.
//{L} ZThread
#include <ctime>
#include "DiningPhilosophers.h"
#include "zthread/ThreadedExecutor.h"
using namespace ZThread;
using namespace std;

int main(int argc, char* argv[]) {
    srand(time(0)); // Seed the random number generator
    int ponder = argc > 1 ? atoi(argv[1]) : 5;
    cout << "Press <ENTER> to quit" << endl;
    enum { SZ = 5 };
    try {
        CountedPtr<Display> d(new Display);
        ThreadedExecutor executor;
        Chopstick c[SZ];
        for(int i = 0; i < SZ; i++) {

```

⑨ 在写本章内容的时候, Cygwin ([www.cygwin.com](http://www.cygwin.com)) 正在进行调整变化, 改善对它的线程处理的支持。利用Cygwin的这个可利用版本下的程序, 我们仍然不能观察死锁行为。举个例子, 该程序在Linux系统下很快地就会死锁。



```

        executor.execute(
            new Philosopher(c[i], c[(i+1) % SZ], d, i, ponder));
    }
    cin.get();
    executor.interrupt();
    executor.wait();
} catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
}
} ///:~

```

注意，**Chopstick**对象不需要内部标识符，而是通过其在数组**c**中的位置来识别它们。在构造**Chopstick**对象时，赋予每个**Philosopher**一个左边和一个右边的**Chopstick**对象的引用，这些是在**Philosopher**可以进餐之前必须获取的餐具。除最后一个**Philosopher**外，将**Philosopher**的座位安排在贴近的一双**Chopstick**对象之间来初始化每个**Philosopher**。最后一个**Philosopher**的右边**Chopstick**是序号为第0根**Chopstick**，这样就完成了环绕桌子的座位摆放。因为最后一个**Philosopher**就座的右边紧挨着第1个**Philosopher**，他们俩共享第0根筷子。这样的安排可能在某一时间点上所有的哲学家同时试图进餐，并且等待紧挨着他们的哲学家放下筷子，这样程序将会死锁。

如果这些线程（哲学家）花费在其他任务（思考）上的时间越多于花费在进餐上的时间，则他们需要共享资源（筷子）时发生冲突的可能性就会越低，因此可以使人相信，这个程序是没有死锁的（使用非0的**ponder**值），尽管事实上它可能会死锁。

为了修正这个问题，必须明白如果同时满足以下4种条件，死锁就会发生：

1) 相互排斥。线程使用的资源至少有一个必须是不可共享的。在这种情况下，一根筷子一次就只能被一个哲学家使用。

2) 至少有一个进程必须持有某一种资源，并且同时等待获得正在被另外的进程所持有的资源。也就是说，要发生死锁一个哲学家必须持有一根筷子并且等待另一根筷子。

3) 不能以抢占的方式剥夺一个进程的资源。所有进程只能把释放资源作为一个正常事件。我们的哲学家是有礼貌的，他们不会从别的哲学家手中抢夺筷子。

4) 出现一个循环等待，一个进程等待另外的进程所持有的资源，而这个被等待的进程又等待另一个进程所持有的资源，以此类推直到某个进程去等待被第1个进程所持有的资源。因此，头尾相接环环相扣，因此大家都被锁住了。在**DeadlockingDiningPhilosophers.cpp**中，是因为每个哲学家都试图先得到右边的筷子，而后再得到左边的筷子，所以发生了循环等待。

因为必须所有这些条件都满足才会引发死锁，那么只需阻止其中一个条件发生就可防止产生死锁。在这个程序中，防止死锁最容易的办法是破坏条件四。这个条件发生的原因是由于每个哲学家都试图以特定的顺序拿筷子：先右后左。正因为如此，才可能进入这样的情形：每个人都把持着其右边的筷子，而等待得到其左边的筷子，由此导致循环等待条件产生。然而，如果最后一个哲学家被初始化为先尝试拿左边的筷子，然后再拿右边的筷子，那么该哲学家将永远无法阻止右边紧挨着的哲学家拿到他自己左边的筷子。在这种情形下，就防止了循环等待。这只是问题的一种解决方法，读者也可以通过阻止其他条件发生来解决该问题（更具体的细节请参考论述高级的线程处理的书籍）：

```

//: C11:FixedDiningPhilosophers.cpp {RunByHand}
// Dining Philosophers without Deadlock.
//{L} ZThread
#include <ctime>
#include "DiningPhilosophers.h"
#include "zthread/ThreadedExecutor.h"

```

```

using namespace ZThread;
using namespace std;

int main(int argc, char* argv[]) {
    srand(time(0)); // Seed the random number generator
    int ponder = argc > 1 ? atoi(argv[1]) : 5;
    cout << "Press <ENTER> to quit" << endl;
    enum { SZ = 5 };
    try {
        CountedPtr<Display> d(new Display);
        ThreadedExecutor executor;
        Chopstick c[SZ];
        for(int i = 0; i < SZ; i++) {
            if(i < (SZ-1))
                executor.execute(
                    new Philosopher(c[i], c[i + 1], d, i, ponder));
            else
                executor.execute(
                    new Philosopher(c[0], c[i], d, i, ponder));
        }
        cin.get();
        executor.interrupt();
        executor.wait();
    } catch(Synchronization_Exception& e) {
        cerr << e.what() << endl;
    }
} ///:~

```

通过确保最后一个哲学家在拿起和放下其右边筷子之前先拿起和放下其左边筷子，就可以消除死锁，程序将会流畅地运行。

没有编程语言上的支持可以帮助防止死锁；这取决于你是否能通过谨慎的设计来避免死锁。这对于那些正试图调试一个发生死锁程序的人来说并不是什么值得安慰的消息。

## 11.9 小结

本章的目标是给读者提供一个采用线程进行并发编程的基础：

- 1) 可以运行多个独立的任务。
- 2) 当这些任务关闭时，必须全面地考虑所有可能的问题。在任务完成之前，它们使用的对象或其他任务可能会消失。
- 3) 任务在彼此争夺共享资源时会产生冲突。互斥锁是用来防止这些冲突的基本工具。
- 4) 如果不谨慎设计的话，任务可能死锁。

然而，有很多其他有关线程处理方面的工具，可以帮助来解决线程处理的问题。ZThread 库就包含有很多这样的工具，比如，信号量（semaphore）和在本章中所见到的与队列相似的特殊类型的队列。可以探究这个库以及其他有关线程处理的专题资源来得到更深入的知识。

至关重要的是要学会什么时候应该使用并发，以及什么时候应该避免使用并发。使用它的主要原因是：

- 为了处理许多任务，这些任务交织在一起，应用并发可以更有效地使用计算机（包括透明地分配任务到多CPU的能力）。
- 为了能够较好地组织代码。
- 为了用户使用更方便。

资源均衡的经典例子是在I/O等待期间使用CPU。给用户带来便利的经典例子是在长时间下载过程期间监视“停止”按钮是否按下。

线程额外的优点是它们提供“轻量级”的执行语境切换（约为100条指令）而非“重量级”进程语境切换（约上千条指令）。由于一个给定的进程中所有的线程共享同一内存空间，一个轻量级语境切换只改变了程序执行的先后顺序和局部变量。进程改变——重量级语境切换——必须调换所有内存空间。

多线程处理的主要缺陷是：

- 当等待共享资源时性能降低。
- 处理线程需要额外的CPU开销。
- 拙劣的程序设计决定会引发毫无益处的复杂性。
- 为诸如饥饿、竞争、死锁和活锁等病态行为的出现创造了机会。
- 跨平台操作造成的不一致性。在为本章开发原始材料（使用Java）时，作者就发现竞争条件在某些计算机上会很快出现，但在另外的计算机上则不会。本章中的C++例子在不同的操作系统下其行为是不同的（但通常是可接受的）。如果在某台计算机上开发一个程序，并且工作似乎一切正常，然而当发布它时你也许会因得到完全不受欢迎的结果而大吃一惊。

与线程一起存在的最大的困难之一是，因为多个线程也许在共享某个资源——比如，一个对象中的内存——并且还要必须确定多个线程不会在同时读取和改变那个资源。这需要头脑精明地使用同步工具，必须要彻底理解这些同步工具，因为它们可以神不知鬼不觉地将程序引入到死锁的境遇。

另外，线程的应用有一定的技巧。C++被设计为允许创建足够多的对象来满足解决问题的需要——至少在理论上是这样。（比如：为进行工程上的有限元分析而创建上百万个对象，这可能是不现实的。）然而，想要创建的线程数目通常有一个上限，因为在达到某个数量时，线程的性能就会变得极差。这个临界点很难探测，且常常依赖于操作系统和线程库；它可以是少于一百个，也可能达到数千个线程。就我们常常只创建少量的线程以解决某个问题而言，这个限制没有多大作用；但是在更一般的设计中它就会变成一个约束。

用一种特定的语言或库来进行线程处理不管似乎有多么简单，都认为它是一种变幻无常的魔术。人们在编程时总会有些没有考虑周全的地方，因此就会在你最没有预料到的时候“咬你一口”。（比如，请注意因为哲学家进餐问题可以进行调整，所以死锁很少发生，人们就会得到一切都万事大吉的假象。）这里引用Python 编程语言的发明者Guido van Rossum的一个恰当的描述：

在任何多线程处理的程序设计中，大多数的错误来自于线程处理问题。这和编程语言无关——它是深层次的问题，即人们至今仍未完全理解的线程的性质。

有关线程处理更高级的讨论，请参看《Parallel and Distributed Programming Using C++》一书，Cameron Hughes和Tracey Hughes著，Addison-Wesley出版社2004年出版。

## 11.10 练习

- 11-1 从Runnable继承一个类，并重写run()函数。在run()中打印一个消息，然后调用sleep()。重复这个过程3遍，然后从run()返回。在构造函数中放进一条启动（start-up）消息，并且在任务结束时打印一个关闭（shut-down）消息。创建几个此类型的线程对象，运行它们并观察会发生什么结果。
- 11-2 修改BasicThreads.cpp，使LiftOff线程启动其他的LiftOff线程。
- 11-3 修改ResponsiveUI.cpp，消除任何可能的竞争条件。（假设bool操作是非原子的。）
- 11-4 在Incrementer.cpp中，修改Count类，使用一个int变量而不使用int数组。解释产生的行为。
- 11-5 在EvenChecker.h中，纠正Generator类中的潜在问题。（假设bool操作是非原子的。）
- 11-6 修改EvenGenerator.cpp，使用interrupt()而不使用退出标志。

- 11-7 在**MutexEvenGenerator.cpp**中, 改变**MutexEvenGenerator::nextValue()**中的代码, 使返回表达式处在**release()**语句之前, 并解释会有什么情形发生。
- 11-8 修改**ResponsiveUI.cpp**, 使用**interrupt()**而非**quitFlag**方法。
- 11-9 查看**ZThread**库中**Singleton**的文档。修改**OrnamentalGarden.cpp**, 以便**Display**对象被一个单件**Singleton**控制, 防止多个**Display**对象被意外地创建。
- 11-10 改变**OrnamentalGarden.cpp**中的**Count::increment()**函数, 使它直接对**count**增1 (即使用**count++**)。现在删去保护, 看看是否会引起失败。这种做法是安全可靠的吗?
- 11-11 修改**OrnamentalGarden.cpp**, 使用**interrupt()**而非**pause()**机制。确保这种解决方案不会过早销毁对象。
- 11-12 修改**WaxOMatic.cpp**, 添加**Process**类的更多实例, 使它对汽车外壳上蜡进行3次上蜡和抛光, 而不是只有一次。
- 11-13 创建**Runnable**的两个子类, 一个子类在**run()**中启动然后调用**wait()**。另一个子类的**run()**获得第1个**Runnable**对象的引用。在若干秒后其**run()**会为第1个线程调用**signal()**, 以使第1个线程可以打印一条消息。
- 11-14 创建一个“忙等待”的例子。一个线程休眠一段时间, 然后把一个标志设置为**true**。第2个线程在一个**while**循环中监视这个标志 (这就是“忙等待”), 当标志变为**true**时, 把它设置回为**false**, 并把这个变化报告给控制台。注意程序在“忙等待”中耗费了多少时间, 创建该程序的第2个版本, 使用**wait()**代替“忙等待”。特别提示: 运行**profiler**, 显示在每种情况下使用的CPU时间。
- 11-15 修改**TQueue.h**, 添加可允许的最大元素计数值。如果元素数达到这个数量, 后续的写操作应该被阻塞, 直到元素计数小于最大元素计数值为止。编写代码检测这种行为。
- 11-16 修改**ToastOMaticMarkII.cpp**, 使用两条独立的流水线, 在烤面包三明治上涂抹花生酱和果酱, 并为已完成的三明治使用一个输出队列**TQueue**。使用一个**Reporter**对象显示结果, 就像在**CarBuilder.cpp**中那样。
- 11-17 使用真正的线程处理而非模拟线程处理重写**Co7:BankTeller.cpp**。
- 11-18 修改**CarBuilder.cpp**, 给机器人添加标识符, 并且添加不同种类机器人的更多实例。注意是否所有机器人都得到利用。
- 11-19 修改**CarBuilder.cpp**, 给汽车制造进程增加另一个阶段, 添加排气系统、车身、挡泥板。如同在第1阶段, 假设这些进程可同时被机器人执行。
- 11-20 修改**CarBuilder.cpp**, 使**Car**可以对所有**bool** (布尔) 变量进行同步访问。因为互斥锁**Mutex**不能被拷贝, 这将需要对整个程序进行重大的修改。
- 11-21 使用本章给出的**CarBuilder.cpp**中的方法, 给房屋建筑的例程建模。
- 11-22 创建一个**Timer**类, 这个类有两个选项: 一个只发射一次的一次射击计时器, 以及每隔一定的时间间隔定期发射的计时器。和**C10:MulticastCommand.cpp**一起使用这个类, 把对**TaskRunner::run()**的调用从程序中移到计时器类中。
- 11-23 改变哲学家进餐的例子中的两处内容, 以便除了思考时间之外, 还在命令行上控制**Philosopher**的数量。尝试输入不同的值并解释结果。
- 11-24 改变**DiningPhilosophers.cpp**, 以便**Philosopher**正确拿起下一根可用的筷子。(当一个**Philosopher**取完其筷子后, 他们把筷子放入一个筷笼中。当**Philosopher**需要进餐时, 他们就从筷笼中取出下两根可用的筷子。) 这种做法能够消除死锁的可能性吗? 能仅通过减少可用筷子的数量而重新引入死锁吗?